AD-A096 339 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
AUTOMATIC RECOVERY IN A REAL-TIME, DISTRIBUTED MULTIPLE MICROPR--ETC(U)
DEC 80 R L ANDERSON

UNCLASSIFIED NL

1 OF 2

AD A
096339

NAVAL POSTGRADUATE SCHOOL

Monterey, California

9 Master's

THESIS

6

AUTOMATIC RECOVERY IN A REAL-TIME, DISTRIBUTED MULTIPLE MICROPROCESSOR COMPUTER SYSTEM

10

by

Richard L. Anderson

11

Dec~~ember~~ 1980

12

156

Thesis Advisor: R. R. Schell

Approved for public release: distribution unlimited

[illegible]

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A096	3. RECIPIENT'S CATALOG NUMBER 339
4. TITLE (and Subtitle) Automatic Recovery in a Real-time, Distributed Multiple Microprocessor Computer System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; December 1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard Lewis Anderson		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December 1980
		13. NUMBER OF PAGES 156
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Fault-tolerance, Automatic Recovery, Reinitialization, Real-time, Kernel, Segmentation, Dynamic Relocation, Dynamic Reconfiguration, Restart

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents an automatic recovery design that supports the fault-tolerant performance of a real-time, distributed, multiple microcomputer system. The recovery mechanism is structured to maintain real-time processing applications where a record of previous computations is not required and data loss is tolerable during the period of recovery. The automatic recovery technique employed is based on system reinitialization in which the system is restored

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

to it's original initialized state and then restarted. The automatic recovery mechanism has been integrated with a hierarchical, distributed operating system which supports a multiprogramming environment. A distinct address space for each system process, that is preserved by the hardware's explicit memory segmentation, in conjunction with the independent kernel and user domains of the operating system are used to facilitate dynamic relocation among identical processor modules. The result is a flexible environment that supports the dynamic reconfiguration of processors and memory during the period of reinitialization.

Automatic Recovery in a Real-time, Distributed, Multiple Microprocessor Computer System

by

Richard Lewis Anderson
Lieutenant, United States Navy
B.S., United States Naval Academy, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1980

Author [illegible]

Approved by: [illegible]
Thesis Advisor

[illegible]
Second Reader

[illegible]
Chairman, Department of Computer Science

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

This thesis presents an automatic recovery design that supports the fault-tolerant performance of a real-time, distributed, multiple microcomputer system. The recovery mechanism is stuctured to maintain real-time processing applications where a record of previous computations is not required and data loss is tolerable during the period of recovery. The automatic recovery technique employed is based on system reinitialization in which the system is restored to it's original initialized state and then restarted. The automatic recovery mechanism has been integrated with a hierarchical, distributed operating system which supports a multiprogramming environment. A distinct address space for each system process, that is preserved by the hardware's explicit memory segmentation, in conjunction with the independent kernel and user domains of the operating system are used to facilitate dynamic relocation among identical processor modules. The result is a flexible environment that supports the dynamic reconfiguration of processors and memory during the period of reinitialization.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to acknowledge and thank my thesis advisor Lt. Col. Roger R. Schell for his encouragement and guidance in this thesis research. His advice and suggestions often provided the needed incentive required to overcome difficult obstacles.

I would like to thank Professor Tien F. Tao and the students and staff of the Naval Postgraduate School Solid State Laboratory. Their assistance in hardware-related areas was invaluable during my research effort.

A special note of appreciation goes to my wife Marianne whose assistance was a significant contribution to this thesis.

I. INTRODUCTION

Automatic fault recovery is the ability of a computing system to continue its specified logical performance after isolating failed physical components. This thesis presents a simple recovery technique that incorporates system reinitialization in a real-time, distributed multiple microcomputer environment. The automatic recovery mechanism is designed specifically to support image processing applications where a record of previous computation is not required. The recovery mechanism uses a dynamic relocation algorithm as a means of reconfiguring the system as reinitialization from a standard initialization state is performed.

The automatic recovery system mechanism, developed by this thesis, is designed for a class of real-time systems in which the loss of a segment of data is tolerable. Because the loss of previous computations are not a dominant factor for recovery in this type of system, automatic fault recovery is simply a task of reinitializing the system and continuing execution.

This thesis uses a flexible initialization mechanism designed by Ross [20] as the basis for an automatic fault recovery scheme based on system reinitialization. The reinitialization algorithm establishes a defined system state (in particular that of the original initialization), with a different physical configuration. After reconfiguration, to eliminate faulty components, the reinitializaton mechanism allows the system to continue the performance of its logical prescribed tasks in a normal manner.

A. FAULT TOLERANCE

Automatic system recovery is part of a broader area entitled fault-tolerance. Although this thesis deals primarily with the concept of system recovery it is necessary to briefly identify and define the other areas that are included under the notion of fault-tolerance. By presenting a picture (or a model) of fault-tolerance, with specific rules relating to individual system requirements, a clear and concise reasoning can be developed for automatic system recovery.

Fault-tolerance is the architectural attribute of a computer system that allows the system to continue it's specific logical tasks when the system's physical components suffer various kinds of failures. A fault-tolerant logic machine is capable of returning from an error state to a state of normal specific behavior thus assuring the survival of the information processing activities. Fault-tolerance consists of three sequential steps:

1. Fault Detection

2. Fault Diagnosis

3. Fault Recovery

Fault detection requires that the existence of a fault be realized. This is accomplished by a detection mechanism that observes some symptoms of the machine that indicates an error has occurred. Fault diagnosis takes place once a fault is detected. The error conditions are analyzed to isolate the fault cause. Steps are then taken to limit the adverse effects on the system and initiate the correct recovery measures. Finally, fault recovery involves specific actions, such as dynamic reconfiguration of the physical components, to secure continued system operation in a normal state or possibly a degraded mode depending on the recovery mechanism implemented.

The presence of fault-tolerance features in a system is a unique attribute. During normal (fault-free) operation fault-tolerance does not provide any performance advantages and in a fault-free machine would be superfluous. With the increase in technical knowledge, computing machines are becoming larger and more complex. As fault-free devices are not a reality the the necessity of fault-tolerance in a computing system becomes more and more apparent. In the fault.prone-physical implementation, fault-tolerance is the insurance of the logic machine against disruptive physical events [1].

B. RECOVERY TECHNIQUES

Recovery techniques are incorporated into systems in order to cope with failures. A failure is an event at which the system does not perform according to specifications. Failures can have numerous causes, but in a computing system, most generally, are the result of either hardware, software or user errors. In order to deal effectively with failures additional components and algorithms must be added to the system. These components and algorithms attempt to ensure that faults , or occurrences of erroneous states, result in limited damage to system computations. Ideally they remove the faults and restore the system to a "correct" state from which normal processing can continue. The additional components and algorithms required in a system to cope with failures are called recovery techniques or mechanisms.

Numerous recovery techniques have been developed, as there are many kinds of failures. The particular recovery mechanism employed in a computer system is dependent on the type of hardware a system uses, the software and data structures involved, system applications and many more important individual system design characteristics. Consideration as to the degree and priority of system recovery is also necessary. Certain systems, such as missile tracking computers, must perform real-time recovery completely to a correct state , while a large data base machine might be required to recover to a previous correct state thus only preserving the data in its files. In an isolated environment, such as an unmanned spacecraft, system recovery techniques might involve graceful degradation. In such a system, failed physical components and the lack of spares may require reconfiguration of the system in order for computation to continue in a degraded mode. Recovery mechanisms also encompass a degree of fault anticipation. Such techniques involve continued recording of data computations, or "checkpointing", in order to have a recent correct state to recover to. Often redundancy plays a large role in recovery techniques where a system with a faulty physical component will simply switch to an identical component which is either performing in parallel or is a backup spare. Many systems, such as nuclear reactor control systems, use a recovery technique that involves just a safe shutdown once a serious fault has been discovered.

No single recovery technique or series of recovery techniques can cope with every possible fault. Many different kinds of recovery procedures have been developed, each technique with its own particular advantages and disadvantages, but each enabling a system to deal effectively with different kinds of failures in different environments.

The recovery techniques considered in the following sections do not encompass all possible schemes of automatic fault recovery and are by no means the only categorization of recovery mechanisms. Instead some of the more widely used techniques are discussed and the kinds of recovery they provide, as related to real-time systems, are briefly described.

1. Backup

Automatic fault recovery incorporating a backup technique is designed to return the system to a previous (presumably correct) state once a fault is detected and diagnosed. To accomplish this task the state of the system is periodically recorded. This recording or "check pointing" provides the most recent correct state of the system and establishes a point from which the system can be restarted and be expected to function normally if all faults have been corrected.

In real-time systems where execution times are critical backup recovery provides a minimum restoration period when program functions are dependent on previous data computations. Additionally checkpointing, in conjunction with a backup recovery mechanism, is applicable in systems where data loss can not be tolerated. Depending on the extent of checkpointing, a copy of critical data can be continually maintained on auxilary storage and restored if necessary using an automatic backup fault recovery technique.

2. Reinitialization

Reinitialization recovery mechanisms are salvation programs [25] that restore the system to a valid state; that of the initialized system immediately prior to its original execution. Reinitialization recovery basically performs backup recovery to a permanently recorded system state (that of the initial system) without any facility for checkpointing. Because no data recording is done reinitialization techniques do not provide for the recovery of data other than that provided during system initialization.

Real-time systems that can tolerate intermittant losses of data are best suited for the recovery technique of reinitialization. Data loss in such a system becomes simply a function of the time required for reinitialization. In applications such as image processing the data loss is tolerable due to large amounts of relatively similar input information and the acceptable disruption in processing due to occasional faults [19].

3. Redundancy

Redundant recovery techniques employ multiple components or modules, to perform the identical task in parallel. The recovery mechanism is initiated if a disagreement occurs between modules at the end of task computation. There are several basic approaches to redundant fault recovery, but all methods essentially involve the substitiution of a faulty module with one that functions properly. Hybrid redundancy [19] is a form of redundant recovery that involves a majority vote of the outputs of several modules. Disagreeing modules are replaced with spares (under control of agreeing modules) automatically. A similar approach termed duplex recovery [19] involves the comparison of the outputs of only two modules. If disagreement occurs diagnostic routines identify the faulty unit and it is replaced or disabled.

The majority of real-time systems developed in the past, and especially those which operate in an isolated environment (no human maintainance available) have employed redundancy to some degree. Redundant systms provide the time response required for time-critical functions and because of their parallel computations data loss is usually not a result. The disadvantages to redundant recovery systems is realized in the overhead required to run identical multiple systems. With the increase in technical knowledge, real-time systems are becoming larger and more complex. The additional effort and expense required to incorporate automatic redundant fault recovery techniques is often not desirable.

4. Graceful Degradation

Graceful degradation, or degraded recovery, returns the system to a fault-free state, but with a reduced computing capacity [1]. Graceful degradation often involves backup recovery or reinitialization to restore the system, but faulty components are not replaced.

Real-time systems, operating in an isolated environment, often employ a form of degraded recovery if spares are not available or have been depleted. This form of recovery, involving reconfiguration of system components, allows a system to continue performing it's normal logical tasks, but usually at a reduced rate. Recovery using graceful degradation can result in the loss of data if the nonreplaceable component is some form of memory.

5. Safe Shutdown

Safe shutdown is the limiting case of graceful degradation [1]. It is carried out when the system computing capacity falls below a minimum acceptable threshold. This form of "recovery" is a fail-safe method that is employed usually as a last resort. Safe shutdown allows a system to be halted before it causes severe damage to components or data and in some cases jeopardizes human life.

The use of a safe shutdown scheme in a real-time system does not provide any significant advantages other than the avoidance of catastrophic consequences in a critical computing situation. Military weapons systems controlled by a real-time system would be an instance where safe shutdown might be employed.

C. MOTIVATION

The Solid State Laboratory at the Naval Postgraduate School is presently conducting research in the area of image processing. Under the direction of Professor T.F. Tao, research and development of "smart sensors" for missile guidance, radar, satellite surveillance and other image processing applications [22] is progressing. The smart sensor platform will require on-board data processing of large quantities of collected image data. To provide the required computing power to process this significantly large amount of data in real-time, a multiple microprocessor system performing asynchronous parallel processing is being developed [2]. To control this computer system an operating system, using the Multics [16] concepts of segmentation in conjunction with Reed's [18] design of virtual processors, has been developed and is presently in the implementation stage. The basic microcomputer operating system design was developed by O'Connell and Richardson [15] and is based on the structure of a hierarchical security kernel. O'Connell and Richardson provided a flexible operating system design that is fundamentally configuration independent and adaptable to a spectrum of systems. The real-time version of this "family" of operating systems was refined and implemented by Wasson [23] and Rapantzikos [17].

One of the primary goals of the Naval Postgraduate School project, directed toward development of a smart sensor platform, is fault-tolerance. Dynamic reconfiguration within a multiple microprocessor computer system, due to periodic maintenance checks or failure of specific components, is the basis for extended performance, if not survival in such a system. The ability of the smart sensor platform to detect faulty processors or memory segments, diagnose the problems and then perform dynamic reconfiguration (if required) and automatic recovery is a necessity for the system in its projected, isolated operating environment.

The operating system design of Wasson is logically organized into a hierarchy that separates the user application processes from the kernel. This modular, layered design lends itself to dynamic reconfiguration where processes can be relocated among physical processors. Additionally the system initialization technique proposed by Ross [20] provides a basis for an automatic recovery mechanism that will reinitialize the system on a new physical configuration after the detection of faulty system components.

D. OBJECTIVES

This thesis is intended to focus primarily on the area of dynamic reconfiguration and automatic recovery of a real-time, distributed, multiprocessor system in a fault-tolerant environment. Using the system initialization mechanism design of Ross [20], as a basis for system reinitialization, and the synchronization primitives developed by Wasson [23] and Rapantzikos [17], for process coordination, this thesis provides an automatic recovery mechanism specifically designed for a real-time, multiprocessor computing system.

Fault-tolerant computer systems in the past have used fault detection and reconfiguration mechanisms which dealt with components at the level of simple devices such as flip-flops and adders. With todays LSI and VLSI technology, it is no longer appropriate to be concerned with such small subunits. The unit of fault detection and reconfiguration should be on the scale of processor/memory [24].

In order to accomplish fault-tolerance functions on the processor/memory scale new methods of detection and recovery have been developed. Software controlled fault-tolerance is a method that has been successfully implemented in such experiental systems as SIFT [24], FTMP [3] and Pluribus [12]. Fault tolerance is accomplished as much as possible by programs in these systems rather than the conventional hardware methods traditionally used. This includes error correction, detection, reconfiguration and prevention of a faulty unit from having an adverse effect on the system as a whole. This modularization (processor/memory) of system components allows fault detection to be based on modular performance. Detection becomes simply an algorithm performed by a system monitor that determines the correct functioning of a module. The monitor evaluation can be performed using various methods. In SIFT [24] a two out of three vote of processor/memory computation determines a faulty module. Recovery techniques in such a system consist of a monitor algorithm that simply eliminates a failed module by marking it as faulty and replaces it with a spare if available. It is the primary objective of this thesis to design a recovery technique that is software controlled. The use of Intel's iSBC 86/12A Single Board Microcomputer with on board RAM provides the processor/memory module configuration necessary for such an algorithm-based recovery mechanism.

Dynamic reconfiguration is usually encompassed in an automatic recovery scheme and essentially involves the automatic reconfiguration of a system in order to eliminate the faulty components. The objective of a modular automatic recovery design, incorporating dynamic reconfiguration, can be realized based on the concepts presented by Schell [21]. The ability to bind and unbind the physical resources to the logical resources of a system creates an environment supportive of dynamic reconfiguration. This in conjunction with an automatic recovery technique, controlled primarily by the system software and designed specifically for a real-time, multiple microcomputer system, is the primary objective of this thesis.

Several designs for system recovery have been developed in recent years. Although specific techniques have been employed, enormous problems still remain to be solved for parallel processors and distributed processing [25]. It is the additional goal of this thesis to provide some solutions to the dilemmas facing fault recovery in parallel processing systems.

The real-time, image processing project under development at the Naval Postgraduate School provides an environment that lends itself to a simple fault recovery technique. Complete system reinitialization after dynamic reconfiguration is a feasible fault recovery method provided the time for system reinitialization does not significantly degrade performance. With the LSI and VLSI technology used in the image processing environment the recovery time will not be a significant factor. Due to the enormous amount of continued input information a few frames not processed during reinitialization will result in only temporary loss of the image and will not significantly degrade performance [2,19].

This thesis deals primarily with only one aspect of fault-tolerance, that of fault recovery. One must assume that fault detection and diagnosis have been performed prior to fault recovery and that the system recovery mechanism has been initiated as a result of a detected fault. It is on these assumptions that this thesis is based.

E. THESIS STRUCTURE

The introduction just presented is designed to provide the reader with a brief look at fault-tolerance as it applies to computer systems and in particular to the development decisions on which an automatic recovery technique is based. Chapter II will describe the hardware architecture of the multiprocrssor system designated for the automatic recovery mechanism and the support utilities that enhance the hardware performance. Chapter III will provide a detailed account of system initialization and how the initialization mechanism was implemented on the system hardware. Chapter IV will outline the automatic recovery design as it relates to the operating system and the hardware employed by the system. The final chapter presents conclusions and observations that resulted from this thesis effort and suggestions for further research. Four appendices are also provided that give detailed descriptions of the system initialization programs and their implementation.

II. SYSTEM STRUCTURE

A. OPEPATING SYSTEM

To use the multiple microprocessor environment effectively for real-time image processing the application programs must be partitioned and distributed among the microprocessors. The operating system required to manage such a multiple microcomputer system must coordinate inter-process communication and synchronization. Additionally the operating system is tasked with the management of system resources which include I/O and memory management.

The distributed operating system designed by Wasson [23] and Rapantzikos [17] supports the multiple microcomputer environment. It provides control for a large number of asynchronous processes and is designed to manage the resources of a multiple microcomputer system. The operating system is structured as a hierarchy, supporting kernel and supervisor domains. Segmentation of memory [16] facilitates the sharing of inter-process data while at the same time isolating the address space of those processes that require no interference. The concept of virtual memory, where each process is provided with its own address space, as supported by segmentation, leads to a configuration independent system.

The kernel manages all physical processor resources providing the user with an environment that is relatively hardware independent while the supervisor provides the interface between the kernel and application processes. Inter-process communication and synchronization is accomplished using eventcounts and sequencers [18] and to ensure expeditious handling of time-critical processing requirements a preemptive, priority scheduling mechanism is incorporated.

The operating system is designed to control a group of multiprocessors which share a single system bus or possibly a set of up to four "clusters" of such microcomputers [22]. In order to limit the bus usage to a minimum, and thus provide increased performance, copies of the kernel are physically distributed to each microprocessor's local memory. This allows for high-speed access to kernel functions without over-burdening the shared system bus.

The distribution of the operating system kernel necessitates its execution by every processor. Thus the kernel design incorporates a scheduler that will allow each CPU to provide its own scheduling. This leads to an operating system that has no concept of master-slave control but, is dependent only on system-wide synchronization variables to maintain system coordination and regulation.

1. The Kernel

The kernel uses the concept of two-level traffic control to manipulate system resources. Multiplexing of the physical processors amongst the more numerous virtual processors is accomplished by the Inner Traffic Controller. It is at this lowest level of the kernel that the hardware of the physical machine is interfaced. At the higher level, the Traffic Controller, virtual processors are multiplexd among the larger number of partitioned application processes. At this upper level of the kernel the inter-process communication and synchronization primitives are made available to the user application processes to solve the complex (application independent) system-wide synchronization of parallel processing.

2. The Supervisor

In the multiple microprocessor operating system family, proposed by O'Connell and Richardson [15], the supervisor level of the system is designed not only to provide the kernel interface, but to support such functions as file management. The modified real-time subset of this operating system family, implemented by Wasson [23] and Rapantzikos [17] for image processing, incorporates the supervisor only as a "gate" to the kernel. The supervisors gate is simply an interface to the kernel for the application process. The gate provides a single entry point to the kernel in which all user programs can access the synchronization primitives. This allows the supervisor level and application processes to be independent of the kernel implementation details and maintains the hierachical design of the system.

3. Real-time Processing

In the isolated environment of the smart sensor platform, real-time processing involves time-critical computations. Real-time systems must be controlled by operating systems that ensure time-critical processing is given immediate attention when required.

The image processing programs of the smart sensor system are partitioned into separate processes and distributed among individual microcomputers. The ability of each processor's kernel to schedule the image processing functions assigned to it is accomplished by a priority-driven preemptive scheduling technique which provides for expeditious handling of processes which perform time-critical operations. Additionally the distribution of the application processes among the physical processors local memories allows the same advantages as the distribution of the kernel. Performance is increased in the real-time environment by reducing system bus accesses for program instructions and data. The placement of all executable code and unshared data in local processor memory enhances the time-critical processing required in a real-time system.

B. HARDWARE

1. Selection

The microprocessor chosen to support the real-time image processing project was the Intel 8086. Significant advantages over comparable microcomputers were realized in the final selection of the 8086 for the multiple microprocessor design. Performance specifications, past experience with other Intel products, and especially the software and peripheral equipment support all added up to an off-the-shelf, immediately available microprocessor that could be easily interfaced to the image processing project.

2. The 8086 Microprocessor

The Intel 8086 is a 16 bit, HMOS technology microprocessor. It has a 5 Megahertz (MHZ) clock rate and can address a full megabyte of primary memory. To provide high execution speed the 8086 architecture incorporates instruction pre-fetch which allows for the overlapping of instruction fetch and instruction execution cycles.

The 8086 uses memory segmentation to divide the one megabyte of accessible memory into logical units. A segment can range anywhere up to 64 kilo-bytes in length and can be placed anywhere within the one megabyte address space of the 8086, provided the segment base begins at a 16 byte boundary [4]. Although segmentation allows for the logical division of memory into an independent set of contiguous locations it must be emphasized that the segment boundry length is not enforced by the hardware. Since the 8086 does not support explicit segment boundries, segments at the hardware level may be disjoint, partially overlapped or fully overlapped. To support the operating system, the design constraints must ensure segments of an individual process never overlap. The mechanisms to achieve this are presented by Ross [20].

To obtain the effective address of a particular memory location the 8086 uses a base address and an offset. The base address must be a multiple of 16. In order to address the full megabyte of memory the 8086 performs a left shift of four bits on the base address, zero-filling the four lower-order bits. Once the base address has been shifted the address offset from the instruction counter register is added to the base value forming a 20-bit effective address.

The 8086 processor has direct access to four segments at any one time [4]. Their base addresses are contained in four segment registers depending on the segment use. The Code Segment (CS) register contains the base address of the code segment from which instructions are fetched. The Instruction Pointer (IP) register provides the offset from the CS value to the next executable instruction. The Stack Segment (SS) register maintains a pointer to the base of the stack segment. The Data Segment (DS) register contains the address of the current data segment and the Extra Segment (ES) register provides an additional segment address that is typically used for external or shared data.

3. The iSBC 86/12A Single Board Microcomputer

The iSBC 86/12A is a complete microcomputer platform [4]. It contains a 5MHZ 8086 processor, 32 kilo-bytes of random-access memory (RAM), 8 kilo-bytes of electrically programmable read-only memory (EPROM), programmable serial and parallel I/O interfaces, a programmable interrupt controller, a real-time clock and an interface to the Intel Multibus for interconnnection to other devices [11].

The iSBC 86/12A provides the basic hardware support required for a multiple processor operating system. The Multibus interface provides each processor with the ability to independently access a global shared memory segment. The 8086 processor provides a built-in semaphore instruction which allows individual CPUs to set a lock on the system bus, and thus control global memory access. The iSBC 86/12A also can be configured to provide preempt interrupts (between processors) by connecting the parallel I/O ports to the Multibus interrupt lines. Finally the EPROM can be programmmed to contain the bootstrap program that will intialize the system.

4. Intel MDS Development System

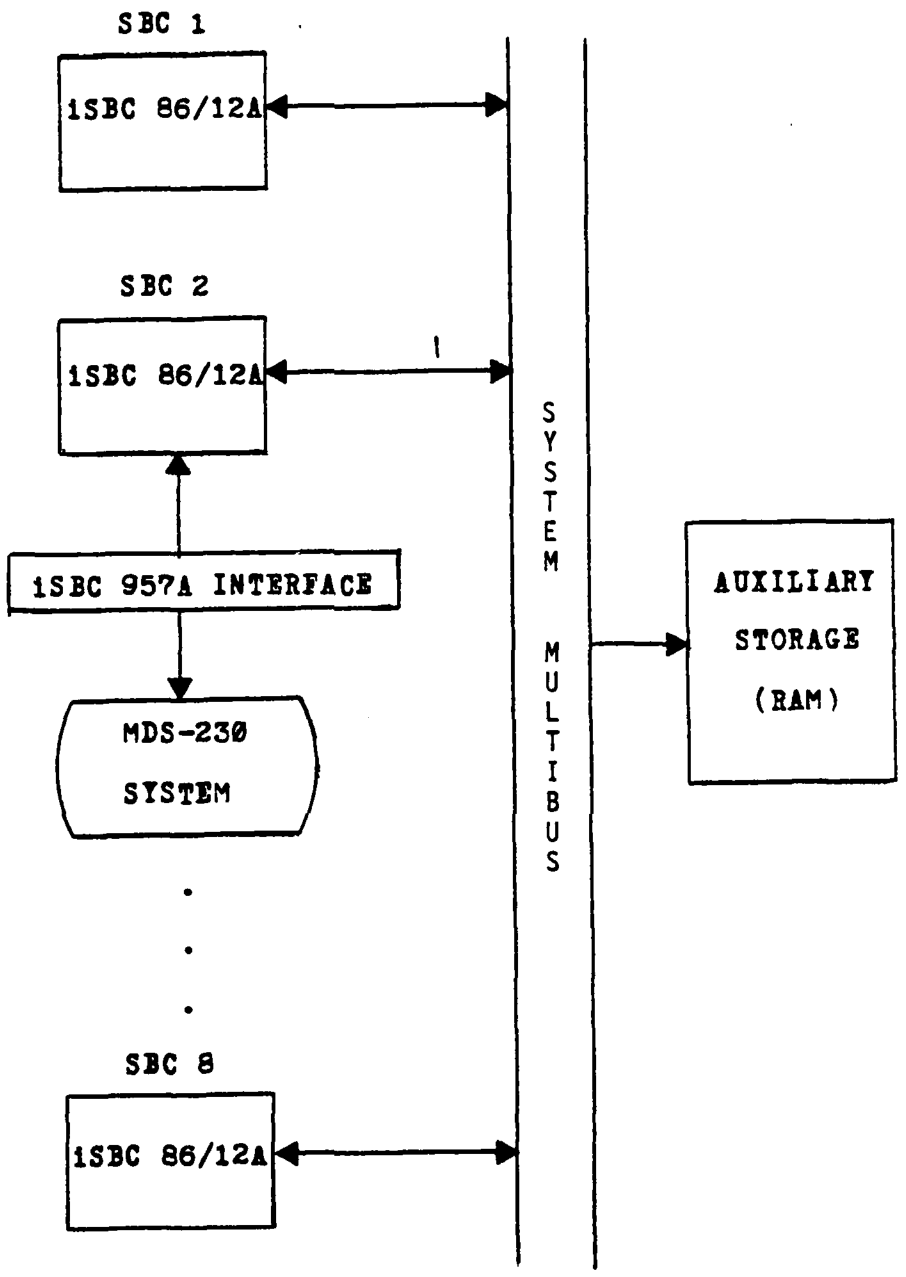
Program development for the real-time multiple microprocessor project was accomplished using the Model 230 Intellec Series II Microcomputer Development System (MDS) [4]. The hardware and software support provided by the MDS was a significant factor in the original choice of Intel's 8086 CPU and iSBC 86/12A single board computer for use in the system.

a. Hardware

Secondary storage for the multiple microcomputer system was not available and therefore the MDS system with its floppy disc file storage, as shown in Figure II-1, was used to simulate secondary storage for the iSBC 86/12As. This was particularly important during system initalization and reinitialization. Since the Multibus was not connected to secondary storage all disc accesses were accomplished through the single iSBC 86/12A connected to the MDS via a serial port link. System I/O was coordinated by a bootstrap program in the case of initialization or by a run-time loader process during system execution. Essentially the iSBC 86/12A connected to the MDS was required to execute a loader process, when disc I/O was required, loading data into a global memory buffer. The other single board processors could then accomplish their individual memory loading by accessing the global memory buffer. It should be noted that this simulation of secondary storage by the MDS is only required until a hard disc is installed and interfaced to the Multibus.

b. Software Utilities

The MDS software support provided by the manufacturer was again one of the prime considerations for



MDS HARDWARE CONFIGURATION

Figure II-1

the selection of the Intel products used in the multiple microcomputer system. The utility programs provided were used extensively in the system generation phase to create the operating system and the initialization programs.

The PL/M-86 compiler [7] provided the necessary support to allow system programming to be accomplished in the flexible, high-level language of PL/M-86 [5]. The language is totally reenterant as reenterant code is essential for the kernel code that is shared by the user processes. The PL/M-86 compiler offered four modes of operation that allowed the programmer to select the degree of segmentation during translation. The compact mode of compiler operation was used primarily during the system generation as it afforded the most flexible use of the segmented address space during process relocation.

The LINK86 [6] utility program was used to combine the separately developed and compiled program modules into a single, relocatable object module. The linking ability provided by this utility routine allowed the programmer to develop small manageable program modules that could be debugged and maintained separately and then bound into a single module prior to loading.

The LOC86 [6] support program produces an absolute object module from the input relocatable object module. This utility routine provides the programmer with the ability to locate object modules at any location in the one megabyte of addressable memory space.

Finally OH86 [6] was used to convert an object module to a hexadecimal, ASCII formatted, object file. This utility program provided formation of an object module in hexadecimal, that could be easily manipulated once loaded into primary memory. The format of the hexadecimal file was such that a simple program within the kernel could read and relocate the object file. The same program of the kernel also converted the hexadecimal module back to a binary object module. This was necessary in order to allow normal execution of the file.

c. The iSBC 957A-iSBC 86/12A Interface

The iSBC 957A Intellec-iSBC 86/12A Interface and Execution Package [9] contains the hardware and software required to interface an iSBC 86/12A Single Board Computer with the Intellec Microcomputer Developement System (MDS). Recall that the system bus (Multibus) that is used by the iSBC 86/12As was not connected to any sort of secondary storage. In order to simulate secondary storage for the system one of the iSBC 86/12As was connected to the MDS and the iSBC 957A interface package I/O routines were used to access the MDS floppy disc drives.

The iSBC 957A interface package contains software utility programs that were used extensively in the research and developement environment of this thesis. The iSBC 957A package system I/O routines interface with the

ISIS-II operating system running on the MDS. The routines can be activated by PL/M-86 high level language procedure calls where the iSBC 957A procedures are declared external in the PL/M-86 program. This allows programs executing in the iSBC 86/12A to perform I/O with the MDS floppy discs. Additionally the iSBC 957A interfaces with the iSBC 86/12A monitor providing the use of the monitor commands for program debugging on the iSBC 86/12A.

An iSBC 957A system I/O procedure is first called in the bootload phase of system initialization. The bootload program calls the routine LOAD [9] to load the bootstrap program, stored on disc, into a buffer in main global memory. This allows all the remaining processors access to the bootstrap routine. The LOAD process requires five parameters to be passed to it. The first argument passed is a pointer to an ASCII string containing the name of the file on disc to be loaded. The next parameter passed to the LOAD routine is a word containing the value of zero; this argument has no effect as it serves only as a placeholder. This parameter is followed by a word that acts as a switch. This argument is set by the programmer and indicates that control be either returned to the calling program or that contol be transferred to the program just loaded. The next argument is a pointer to a pointer in which the starting address of the loaded program is placed. The final argument passed to LOAD is a pointer to a word in which the monitor can place a status code indicating a nonfatal error has occurred during the LOAD routine.

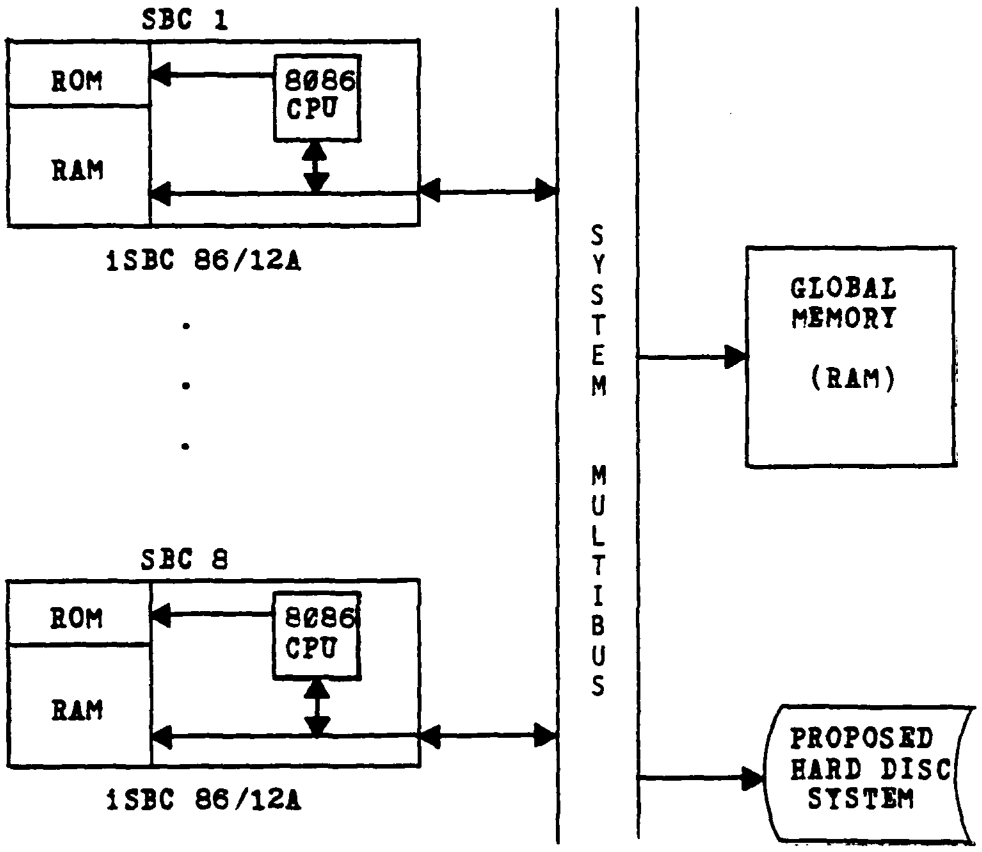
The iSBC 957A system I/O procedures are also used in the bootstrap process of system initialization. During the bootstrap program the OPEN, READ and CLOSE [9] routines are called to read a hexadecimal object file containing the base layer of the operating system into a buffer in global primary memory. The OPEN procedure locates the specified file to be read, on disc, and then initializes ISIS-II tables and buffers in the Intellec system. Five parameters are passed to the OPEN routine. The first argument is a pointer to a word in which the monitor stores the active file transfer number (AFTN). This number is used to identify the file to other iSBC 957A system I/O procedures. The next parameter is a pointer to an ASCII string containing the file name. Following the pointer to the file name is a word containing the access mode for which the file is being opened. This argument identifies the file attribute as read, write or read and write. The next parameter is a word containing a file number that is used only if line editing is taking place (this argument was not used). The final argument is a pointer to a word in which the monitor could pass a status code if a nonfatal error occurred during the OPEN routine.

The READ procedure is called by a PL/M-86 program to transfer up to 4096 bytes of data from an open file to a memory location specified by the calling program. The first argument passed to READ is a word containing the active file transfer number (this will be the same file number assigned in the open procedure, if OPEN and READ are used in conjunction). The next parameter is a pointer to a buffer to which data of the open file is to be transferred. A word containing the number of bytes to be transferred is the next paramenter passed to READ. This argument is followed by a pointer to a word in which the actual number of bytes transferred is placed upon completion of the READ procedure. The final argument passed to READ is a pointer to a word in which the monitor will return a status code in event of a nonfatal error during READ routine.

A call to the CLOSE procedure will cause the ISIS-II operating system to delete the tables and buffers that were allocated when the specified file was opened. The arguments that are passed to CLOSE include the word containing the active file number (the same as assigned in OPEN) and a pointer to a word in which the monitor can return a status code should a nonfatal error occur during the CLOSE routine.

The only other iSBC 957A procedure used was the EXIT [9] routine. This procedure allowed a PL/M-86 program executing on the iSBC 86/12A to return to the monitor if it was called. The EXIT routine was used only for program development and debugging.

Although the iSBC 957A system I/O routines were also used in the run-time loader process to load the application processes and by the loader process in the operating system for system reinitialization it must be emphasized that the iSBC 957A package was used only to simulate an environment. The lack of a hard disc for system secondary storage necessitated the use of the iSBC 957A software and hardware to simulate the required auxilary storage. Future plans for system design (see Figure II-2) include the connection of a hard disc to the Multibus for secondary storage. When this occurs the simulated environment will be eliminated as will be the requirement for the iSBC 957A-iSBC 86/12A Interface and Execution Package.



PROPOSED SYSTEM CONFIGURATION

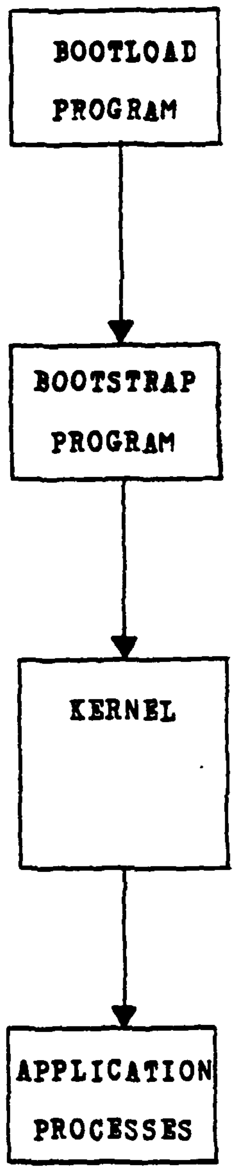
Figure II-2

III. SYSTEM INITIALIZATION

A. DESIGN

System initialization is the method used to get an operating system loaded and running on a computer system. A simple system initialization mechanism has been designed by Ross [20] that can be used with a variety of hardware and operating system configurations. During system initalization Ross outlined three phases that must be accomplished, sequentially, in order to get an operating system loaded and running on a computer system. First, a core image of the operating system is created. This is known as system generation time. It normally is done on a separate development computer system and consists primarily of developing the operating system and initialization code. The next phase of initialization is bootload time. This is the point where the lowest level of the operating system is actually loaded into the primary memory and its system parameters and tables are initialized. Finally when the operating system programs are running normally the initialization sequence is considered to have entered the run time phase.

The initialization mechanism involves three separate loading functions as shown in Figure III-1. The bootload program runs on bare system hardware, during bootload time,



INITIALIZATION SEQUENCE

Figure III-1

and is used to load into global memory a bootstrap program. This program is ROM-resident so that it may be activated by a "bootload" switch. The bootstrap program, loaded by the bootload program, also runs on the bare system hardware and will be used to load the base layer of the operating system into primary memory and start it running. The final loading function is part of the distributed operating system and is loaded into each processor during the bootload phase along with the base layer of the operating system. This loader is used during run time to load the remainder of the operating system and the application programs and to prepare them to be scheduled and run.

Implementation of Ross' system initialization design was the first effort of this thesis with the premise that the initialization technique would be the basis for system reinitialization. This section deals primarily with the specific implementation of the initialization design as it applies to the operating system of Wasson [23] and Rapantzikos [17] and the Intel iSBC 86/12A Single Board Microcomputer.

B. SYSTEM GENERATION TIME

The development of the operating system and initialization tasks takes place at system generation time. This is the first step of initialization and takes place prior to the bootload and execution phases. Program development during system generation was accomplished almost entirely on the Intel Microcomputer Development System (MDS). The use of the ISIS-II operating system in the MDS system with, its supportive utility programs, provided a flexible environment in which to accomplish system generation tasks. The complexity of the bootload and run-time phases was significantly reduced by use of the MDS, in conjunction with the ISIS-II operating system, to compile, link, locate and debug programs during the system generation phase.

In the initialization design by Ross [20], several assumptions were made at system generation time that greatly simplified bootload and run time development. Although some of these assumptions will not hold in the following chapters concerning automatic recovery techniques, for the purpose of system initialization alone this discussion will make the same initial assumptions that Ross does. These assumptions permit extensive preliminary processing to be done in the more flexible atmosphere of system generation thus relieving later phases, which occur in much less supportive environments, of the preparatory processing that they would otherwise be required to perform.

The key assumption at system generation time is that the initial hardware and software configurations are known. This allows initial memory allocation decisions to be accomplished (prior to loading and execution) in the supportive atmosphere of the Intel MDS. The significance of knowing the initial configuration is realized in the ability of the system developer to allocate memory on a global or local scale. As was pointed out in the section describing the operating system, it is highly desirable to place as many programs in local memory as possible in order to eliminate bus contention. Only shared, writable segments should be allocated to global memory.

System generation is viewed as a sequence of events, beginning with program design and ending with the creation of the load module or core image to be loaded. This thesis will concentrate on the specific implementation considerations of the initialization scheme rather than the design methodology. A detailed examination of system generation events and the choices made throughout the development of the initialization design is discussed by Ross [20].

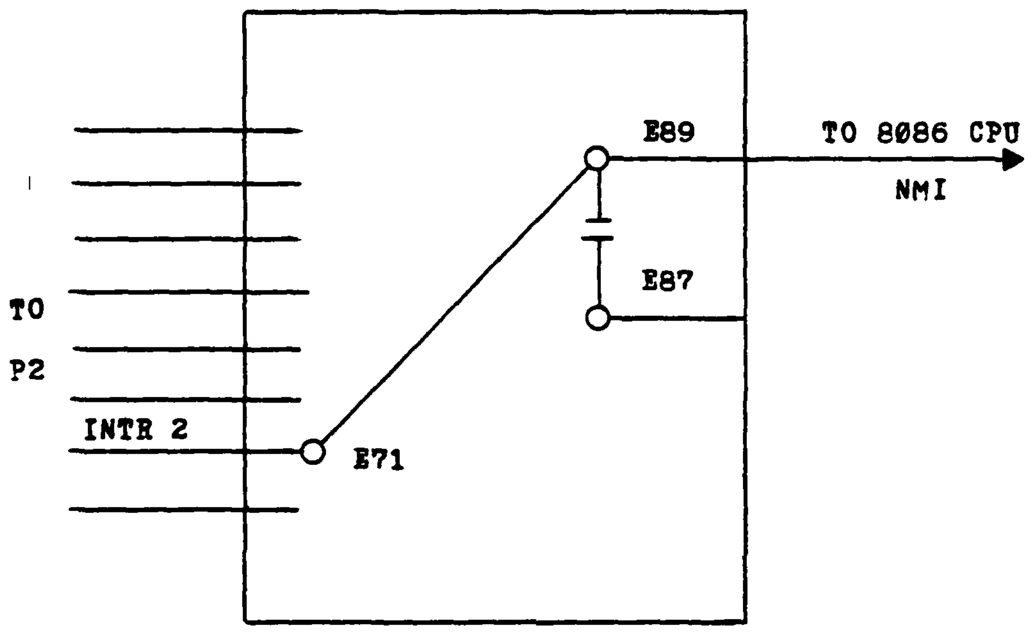
C. BOOTLOAD TIME

The system initialization mechanism was designed to commence operating once a "bootload switch" was activated. This in turn causes a jump to the first instruction of the bootload program which is contained in read-only memory (ROM). The bootload program is a small simple program that runs on the bare hardware and is located in each microcomputer's ROM. The bootload program serves two purposes. It's primary function is to load a "bootstrap" program from secondary storage (i.e., a hard disk) which will then be executed to continue the majority of system initialization. Proceeding in this fashion allows the ROM-resident bootload program to remain small and relatively simple. Secondly the bootload program serves to uniquely identify each physical processor. Each microcomputer's copy of the bootload program differs only in that it contains a unique serial number that identifies the physical processor. This unique processor number is placed in a global CPU table, during execution of the bootload program, and will be used by the bootstrap program to identify the physical processors during the remaining phases of system initialization.

A time-sequence of activities takes place during bootload time, beginning when the bootload switch is pressed, and ending when the operating system kernel is loaded and running. In this particular system the operating system, as was described previously, is distributed to each single board computer and therefore must be loaded into each computer's local memory. Therefore, each microcomputer's bootload program must be activated as it is the responsibility of each individual CPU to load its own system programs. Activation of all the processor bootload programs can be accomplished simultaneously using a simple bootload switch that is connected to all CPUs.

1. System Activation

In the implementation described by this thesis, using one to eight iSBC 86/12A single board microcomputers, it is necessary to indicate to every iSBC 86/12A when to begin executing the ROM bootload program. This was accomplished during development in the form of a simulated bootload switch. In the experimental environment the INTR button on the iCS 80 Chassis [10] served to simulate the bootload switch. Depressing this button places a hardware interrupt on the system Multibus which can be received by all iSBC 86/12As plugged into the iCS 80 Chassis. Interrupt number two is the Multibus interrupt line activated by pressing the INTR button. All iSBC 86/12As can be jumpered to acknowledge this interrupt by wiring the incoming Multibus interrupt line (post E71) to the 8086 non-maskable interrupt line in the interrupt matrix (post E89) [11]. Note that to make the non-maskable interrupt active, the ground wire (between post E87 and E89) must be disconnected. Figure III-2 shows the correct wiring. The non-maskable interrupt on the 8086 has been used to start the system initialization mechanism due to the disabling of the maskable interrupts when the iSBC 86/12A is in the monitor. The initialization routine commences with all boards, except the MDS-connected iSBC 86/12A (as noted below), in their respective monitors. Only the non-maskable interrupt is capable of interrupting iSBC 86/12A INTERRUPT MATRIX



NON-MASKABLE INTERRUPT WIRING

Figure III-2

the 8086 CPU in this state.

When all iSBC 86/12A boards have their interrupt matrix modified as outlined above it is possible to commence the bootload phase, causing all iSBC 86/12A's to execute the bootload program, load the operating system kernel, and commence kernel execution, by simply pushing the INTR button on the iCS 80 Chassis. The bootload program is the interrupt handler. The four byte non-maskable interrupt vector, that will be loaded with the address of the entry point to the bootload program, is the third interrupt vector in the interrupt table [4] (interrupt 2; address 0000:0008 to 0000:000B). Activation of the non-maskable interrupt on the 8086 causes an unconditional, indirect jump to the bootload program via the non-maskable interrupt vector.

System design calls for the bootload program to be ROM-resident, but to facilitate debugging in the experimental environment, it was located in RAM. During this development period the iSBC 86/12A monitor command, LOAD [9], was utilized to download the bootload program from the MDS floppy disc prior to activation of the initialization mechanism. Recall that only one iSBC 86/12A was connected to the MDS in this simulated environment, thus allowing only that particular single board computer to be loaded using the monitor LOAD command. This in turn, required that the bootload program, once loaded, be placed in all the remaining iSBC 86/12As by the monitor MOVE [9] command as it was impossible to load the individual iSBC 86/12A's memories directly. Additionally, all interrupt vectors were required to be preset to the bootload program entry address before the initialization routine could be activated.

Finally the MDS-connected iSBC 86/12A was required to have exited it's monitor before the non-maskable interrupt would function properly. This requirement was the result of MDS interference during the interrupt sequence. To free the iSBC 86/12A, connected to the MDS, of it's monitor it was necessary to start the 8086 CPU executing instructions from RAM. The program executed for this purpose was in the form of a loop at the beginning of the bootload module. When interrupted the CPU then functions identically to the remaining processors. Note that all the other iSBC 86/12As were interrupted while in their respective monitors and functioned normally, thus they required no looping mechanism.

It is necessary to emphasize that the above sequence of events is required only in the experimental environment when placing the bootload program in RAM. When the debugged, final version of the bootload program is located in EPROM the steps involved above will not be applicable.

2. The ROM-resident Bootload Program

The bootload routine is a small, simple program that will be EPROM resident (see Appendix B). The first function of the bootload process is to determine the "Bootload CPU".

The Bootload CPU will serve as the master or controlling CPU throughout the bootload and run time loading phases. While the bootload programs in all CPUs are identical, the Bootload CPU will execute some sequences of instructions that the other processors will not. The PL/M-86 language provides a built in procedure known as Lockset [5] that permits to programmer to implement *a software lock* (*viz.*, *a* busy wait). This procedure uses a variable located in global memory to control the bus access. In order to designate the Bootload CPU, a deliberate race condition is entered as all processors begin execution of the bootload program. Each CPU attempts to set a software lock, using a global variable (CPU$TBL$LOCK), and then enter a table in global memory known as the CPU Table (CPU$TABLE), shown in Figure III-3. The built in procedure Lockset with it's global parameter (CPU$TBL$LOCK) is used to resolve the conflict of multiple simultaneous access attempts to the CPU Table. Thus only one CPU at a time can access the CPU Table and the first CPU to do so becomes the Bootload CPU.

After entering the CPU Table (CPU$TABLE) each processor will fill in entries in the table and then unlock the bus to allow the other CPUs access. The CPU Table is indexed according to logical CPU numbers where the Bootload CPU is designated Ø. The next CPU to get control of the bus, INDEX BY LOGICAL CPU ID ↓

CPU$ID	CPU$ACK	CPU$MAIL	CPU$TOTAL

CPU TABLE

Figure III-3

after the Bootload CPU, and enter the CPU Table, becomes logical CPU 1 and so on.

Once a processor has gained control of the bus using the global bus lock variable (CPU$TBL$LOCK), and accessed the CPU Table (CPU$TABLE) the first action performed is for the CPU to enter its serial number (CPU$ID). Recall that this serial number is different for each ROM-resident bootload program and that this number uniquely identifies every physical processor in the system. Next a counter, (CPU$TOTAL), is incremented in order for the Bootload CPU to keep track of the number of physical processors present in the system. Each CPU is identified additionally by a logical CPU number, (LOG$CPU$ID), that identifies it, as mentioned before, according to its sequence of entry into the CPU Table. The next set of instructions executed in the bootload program increments a logical CPU number (LOG$CPU$NUM). This global variable will be used by the next processor, to gain access to the CPU Table, and will serve as an index into the CPU Table. Finally the software lock on the system bus is released and the identical sequence of entries into the CPU Table is performed by the next processor to gain access to the bus. This continues until all physical processors have accessed the CPU Table and made the appropriate entries. Upon completion the CPU Table (CPU$TABLE) will contain each individual processors unique serial number (CPU$ID) entered according to the sequence of CPU Table access. This allows the processor to be identified by a logical, as well as a physical, CPU number. Additionally the Bootload CPU will have recorded the total physical CPUs it counted in the system in it's own CPU total (CPU$TOTAL) field in the CPU Table. Note that the CPU Table contains a mailbox (CPU$MAIL) entry and an acknowledgement (CPU$ACK) entry for each processor. These entries in the CPU Table will be used later in the bootstrap program for system synchronization.

After completion of the above sequence the Bootload CPU will execute another PL/M-86 built-in procedure called TIME [5]. This untyped procedure causes a time delay in multiples of 100 microseconds based on a 5 MHZ clock and the 8086 CPU cycle time, without interruptions. In the bootload program the Bootload CPU will execute a time delay of 10 milliseconds. This delay will allow all the other processors the time necessary to access the CPU Table before the bootload CPU commences its actual loading action.

The hardware configuration for system development, as described in the hardware section, allows for only one iSBC 86/12A to be connected to the MDS (using the iSBC 957A-iSBC 86/12A interface and execution package). This means that only the single board CPU with this connection can access the disc files. This simplifies the bootload programs by eliminating the need for a complex synchronization method to allow the processors to share the disc, but neccessitates a controlling or Bootload CPU to serve as the main access to disc files for all CPU's. Because the Intel hardware dictates this particular configuration, it is necessary to designate the 86/12A single board microcomputer connected to the MDS, and thus the disc files, as the "Bootload CPU". In order to default the particular processor with the MDS connection as the Bootload CPU a time delay has been added to the instructions of the bootload procedure, BOOTLOAD$INTR (in the bootload program), of all CPU's except the MDS connected iSBC 86/12A. This added time delay in all the processsors, except the Bootload CPU, is executed as the first instruction upon entering the bootload program, thus allowing the iSBC 86/12A connected to the MDS to access the CPU Table (CPU$TABLE) first and become the Bootload CPU. It should be emphasized that this and the unique physical CPU number are the only difference in the bootload programs loaded to the various physical processors and is dependent on the hardware configuration. Note that with a hard disc, serving as secondary storage, connected directly to the Multibus (i.e., all processors are capable of disc access) the need for the default delay will be eliminated as any CPU can serve as the Bootload CPU.

3. Bootstrap Program Loading

The next function of the bootload program is to load a bootstrap program. The bootstrap program (see Appendix C) contains the actual instructions that will load the base layer of the operating system. By performing the initialization in this sequence, the bootload routine remains small and the primary goal, of a simple EPROM resident bootload program is achieved.

The hardware configuration, as described in the previous section, allows for only one iSBC 86/12A to be connected to the MDS and necessitiates this CPU to be the Bootload CPU. Because the Bootload CPU is the only processor that can access the disc files, it must load the files containing the Bootstrap program and the operating system into global memory buffers and then allow the other individual CPU's to execute or load the files as required.

The bootstrap program is loaded by the Bootload CPU using a 957A I/O procedure called LOAD [9]. As was previously described in the hardware section, this utility procedure requires that five parameters be passed to it. The first argument is a pointer to an ASCII string of the file name of the file to be loaded. In this case the bootstrap program (BTSTRP). The next parameter, known as the bias, is not used for this implementation. Following this is a parameter called the switch. This is set to allow the LOAD procedure to return to the bootload program. The next argument is a pointer to the starting address of the loaded program (BTSTRP) which is assigned to the variable ST$BTSTRP$ADR. The last pramenter passed is a status variable for error codes. The Bootstrap program's location in global memory is predetermined at system generation thus the bootstrap program loaded using the iSBC 957A LOAD procedure is a file created by LOC86 which is in executable format (viz., not a hexadecimal file.)

Having successfully loaded the Bootstrap program into global memory the Bootload CPU will transfer control, with an unconditional jump, to the starting address of the Bootstrap program. This transfer of control takes place using a PL/M-86 Indirect Procedure Activation [5] (i.e., simply a call with a pointer). The iSBC 957A LOAD procedure automatically placed the start of the bootstrap program in the start address parameter (ST$BTSTRP$ADR) when it loaded the Bootstrap program. The call, using this bootstrap start address (ST$BTSTRP$ADR), simply sets the CS and IP registers of the Bootload CPU to the starting address of the bootstrap program, puts the parameters to be passed, LOG$CPU$ID, the address of CPU$TABLE and the address of CPU$TBL$LOCK, on the stack and then executes an unconditional jump. This transfers control from the EPROM bootload program in the Bootlaod CPU to the bootstrap program just read in from disc.

While the Bootload CPU is executing the instructions to load the bootstrap program, the remaining processors must enter a wait state. Since the bootload programs are executing on bare hardware the operating system synchronization mechanisms are not available. The solution to CPU synchronization has been to implement a software spinlock in the EPROM resident bootload program called CPU$WAIT. This procedure allows all CPU's except the Bootload CPU to wait in the Bootload program until they are instructed by the Bootload CPU to transfer control to the bootstrap program. The indication for a particular CPU to jump to the bootstrap program, as the Bootload CPU did with a pointer call, will be the placement of the bootstrap start address in the CPU's mail box. Once the processor sees it's mailbox no longer contains the initialized null value it will transfer control from its own EPROM bootload program to the bootstrap program. Note that the bus lock must be set each time a particular CPU accesses The CPU Table (CPU$TABLE), in the spinlock procedure CPU$WAIT, and then released when the CPU exits. This allows the spinlock to function normally in all CPU's with every processor getting a chance to check its mailbox periodically. If this weren't the case one CPU could lock the bus and enter a permanent wait state (in CPU$WAIT). With the bus locked the Bootload CPU would be unable to gain access to the CPU Table (CPU$TABLE) to signal the processor in the CPU$WAIT procedure to transfer control to the bootstrap program. The result would be a deadlock condition.

4. Bootstrap Program Execution

The bootstrap program, created at system generation time, will load the base layer (kernel) of the operating system from disc into primary memory (see Appendix B). As outlined in the previous discussion concerning the operating system, the kernel will be distributed to all physical processors and thus each processor will need to execute the bootstrap program to load it's individual kernel. The Bootload CPU, now executing in the bootstrap program will coordinate the kernel loading among processors and will also do the actual disc access for all CPUs.

The actual entry point to the bootstrap module is the procedure BOOT$STRAP. Since the bootstrap program is not linked to the bootload program the address of the procedure BOOT$STRAP must be the start of the bootstrap module. The entry point must be a procedure as the transfer of control from the bootload program to the boostrap program is a procedure call (ie., call by pointer) which passes parameters. The parameters passed are required by the Bootload CPU to maintain control of the initialization in the bootstrap program. The parameter LOG$CPU$ID identifies each processor as it enters the bootstrap program. The parameters containing the address of CPU$TABLE and CPU$TBL$LOCK (pointers) are used to address based variables [5], CPU$TABLE and CPU$TBL$LOCK, which function identically as they did in the bootload program.

The first action of the Bootload CPU, in executing the bootstrap program, will be to read into a global memory buffer (KERNEL$BUFFER) the hexadecimal file containing the base layer of the kernel. This is accomplished using, as was previosly described in the hardware section, the iSBC 957A Interface Package System I/O procedures [9] in conjunction with the ISIS-II operating system. The first procedure called is OPEN [9]. This procedure essentially locates the kernel file on disc and assigns to it an active file transfer number (KERNEL$AFTN). The next ISBC 957A procedure called is READ [9]. This routine identifies the open file by its active file transfer number (KERNEL$AFTN) and then reads a maximum of 4096 bytes from disc to the global memory buffer (KERNEL$BUFFER). After doing so READ returns the number of bytes transferred in the word TRANS and updates a file marker according to the number of bytes actually transferred. The Bootload CPU will continue to execute the iSBC 957A READ procedure in the bootstrap program until the bytes transferred are less than the maximum bytes allowed for transfer (4096) indicating the end of file has been read and loaded into the kernel buffer (KERNEL$BUFFER). Finally the procedure CLOSE [9] is called allowing the ISIS-II operating system to perform the actions necessary to close the file with the previously assigned active file transfer number (KERNEL$AFTN).

The kernel file just read into the kernel buffer (KERNEL$BUFFER), by the Bootload CPU, is a hexadecimal file created during system generation time by OH86 [6]. When the kernel file is transferred to the kernel buffer it remains in its hexadecimal format. The procedure READ$HEX$FILE will convert the hexadecimal object file (the kernel) into its binary (executable) representation and load it at the address specified in the hexadecimal file. READ$HEX$FILE is executed by the target CPU to load the kernel into it's local memory after being signalled to do so by the Bootload CPU. This method of loading the kernel file as a hexadecimal file was used due to the documentation available, by Intel, with respect to hexadecimal data records. Ross [20] also provides a detailed explaination of hexadecimal record format. Documentation concerning binary object files was less clear than the hexadecimal documentation and did not provide for easy relocation during the bootstrap loading sequence.

Since the Bootload CPU was the first processor to transfer control to the bootstrap program and is the only processor executing in the bootstrap program at this point, it calls the procedure READ$HEX$FILE as soon as it has completed loading the kernel file and passes to it the address of KERNEL$BUFFER. READ$HEX$FILE now loads the kernel file located in global memory into the local memory of the Bootload CPU. Note that the location of the kernel file in local memory is determined at system generation time.

All other processors are still executing the EPROM bootload program, waiting to be signalled by the Bootload CPU via their respective "mailboxes". The Bootload CPU will determine the number of remaining processors waiting to load the kernel file by setting the Bootload CPU (logically 0) processor count equal to the total CPUs (TOTAL$CPUS) minus one (the Bootload CPU doesn't count itself). The Bootload CPU now signals each CPU in turn to load its kernel (converting hexadecimal to object) and then waits in a spinlock until that particular processor has completed that portion of the bootstrap program that loads the kernel into local CPU memory. The signal placed in the target CPUs mailbox is just a pointer to the procedure BOOT$STRAP (in global RAM) which allows the target processor to identify the start of the bootstrap program and transfer control to that address with a pointer call.

The system initialization mechanism is designed to handle kernel files that differ according to individual CPU's assigned functions. For this reason the Bootload CPU allows only one CPU to load the kernel at a time. This allows the Bootload CPU to check which CPU a particular kernel is targeted for and then send the appropriate signal for loading. If the kernel loaded for all processors was identical then the Bootload CPU could signal all the remaining CPUs, simultaneously, and the loading of the kernel could proceed in parallel. Note that in the particular implementation used for development by this thesis the kernel loaded was identical for all CPUs, but the loading was accomplished sequentially to remain consistant with the overall design.

As in the bootload program the bootstrap routine is executing on bare hardware and thus no synchronization mechanisms are available for process coordination. To provide process synchronization a spinlock identical to that used in the bootload program was implemented. The procedure WAIT$CPU allows the Bootload CPU to enter a wait state after signalling a particular processor to transfer to the bootstrap program and load its kernel. When the target CPU has completed loading its kernel it signals the Bootload CPU via the acknowledge flag (CPU$ACK) in the CPU Table (CPU$TABLE). The Bootload CPU then continues to the next logical CPU and repeats the signalling action until all processors, as indicated by the total CPU count (TOTAL$CPUS), have loaded their respective kernels.

As each processor completes its bootloading task it will enter a wait state by calling the procedure CPU$WAIT. Each CPU will remain in this wait state, executing a spinlock, until all processors have completed their respective bootloading tasks. When the loading of the kernel file has been completed by all processors the Bootload CPU will signal all CPUs to perform an unconditional jump to the start location in their respective kernels. This is accomplished by the Bootload CPU setting the acknowledge flag (CPU$ACK) for the Bootload CPU in the CPU Table (CPU$TABLE).

Since the kernel is not linked to the bootstrap program the transfer of control from the bootstrap program to the kernel is accomplished by an indirect procedure activation (viz., a call by pointer). During the previous execution by all CPUs of the procedure READ$HEXFILE, where a kernel was loaded into each CPU's individual local memory, the Code Segment (CS) and Instruction offset (IP) were obtained for each individual kernel. The CS and IP constitute the entry point (start address) of each particular CPU's kernel.

A bootstrap pointer variable (MEM$KCSIP$PTR) is employed using the PL/M-86 language AT attribute [5] to perform the necessary transfer of control to the kernel. The AT attribute locates a two word structure (KCSIP) at the address of the pointer variable (MEM$KCSIP$PTR). Effectively this allows the four byte location in memory reserved for the pointer variable (MEM$KCSIP$PTR) to be accessed two bytes (a word) at a time. Immediately prior to the call by pointer (using MEM$KCSIP$PTR) the first word, of the two word structure, (KCSIP.SEG) is set equal to the kernel code segment (CS) that was determined by the procedure READ$HEX$FILE. The second word (KCSIP.OFF) is set to reflect the kernel instruction pointer (IP). Since the two word structure (KCSIP) uses the identical location in memory as the bootstrap pointer variable (MEM$KCSIP$PTR) the result is to establish the kernel entry point in the bootstrap pointer variable. This allows a pointer call (using MEM$KCSIP$PTR) to transfer control from the bootstrap program to the start of the kernel module.

The pointer call will also pass parameters to the kernel, In particular the logical CPU identification (LOG$CPU$ID) and the physical CPU identification (PHYS$CPU$ID). These arguments are required by the kernel processes in order to identify individual processors. The transfer of control to the kernel is executed by all processors, including the Bootload CPU, after the Bootload CPU has signalled that the loading of the kernel is complete.

It is necessary to keep all processors in a wait state in the bootstrap program and transfer control to the kernel in mass. Should CPUs be allowed to jump directly to their particular kernels immediately after completion of kernel loading, but prior to completion of kernel loading by all CPUs, the global shared variables used by the kernel could be, and most probably would be, altered. These shared variables are "loaded" as part of each kernel, and therefore, would revert to their initialized values. The global shared kernel variables provide for process synchronization and inter-communication and require the presence of all CPUs and respective processes, assigned at system generation time, to function correctly. Allowing processors to transfer intermittently to their kernels would lead to improper initialization of the operating system and erroneous execution.

D. RUN TIME

The transfer of control from the bootstrap program to the kernel, by each physical processor in the system, marks the termination of the bootload phase and the start of the run-time phase of system initialization. During run-time all the user's application processes will be loaded from auxiliary storage by a kernel process called the run-time loader. Unlike the bootload and bootstrap programs, that were required to execute on the bare hardware of the system, the run-time loader will be supported by the kernel functions to facilitate synchronization during the loading of the application programs.

1. The Kernel Interface

The entry into the kernel requires that the parameters passed from the bootstrap program (LOG$CPU$ID and PHYS$CPU$ID) be removed from the stack and that the environment of the kernel be established to ensure proper performance of the operating system. This is accomplished by a special kernel interface set of instructions called the intialization sequence (see Figure III-4) that is located in the Inner Traffic Controller (ITC) Scheduler module [23] of the kernel.

To simplify the transfer of control the entry point

```
 ; FILE SKED.ITC

 ;ESTABLISH STACK STRUCTURE FOR PASSED
 ;PARAMETERS FROM THE BOOTSTRAP PROGRAM
 STACK-STRUCTURE STRUC
    RETURN    DD  ?
    PARM2     DB  ?
    XXX2      DB  ?
    PARM1     DB  ?
    XXX1      DB  ?
 STACK-STRUCK ENDS

 PRDS SEGMENT EXTERNAL
    ;RESERVE MEMORY IN THE KERNEL FOR THE
    ;PARAMETERS PASSED FROM THE BOOTSTRAP
    ;PROGRAM
    LOGCPUID  DB  ?
    PHYSCPUID DB  ?
 PRDS ENDS

;BEGIN THE ITC SCHEDULER SEGMENT IN THE KERNEL
SCHEDULER SEGMENT

   ;BEGIN THE KERNEL INITIALIZATION SEQUENCE
   ;ESTABLISH THE BASE OF THE STACK-STRUCTURE
   MOV   BP,SP

   ;SET UP STACK USING BP AS A BASE POINTER AND
   ;STORE THE PARAMETERS PASSED FROM THE BOOTSTRAP
   ;PROGRAM
   MOV   CL,[BP].PARM1
   MOV   ES:LOGCPUID,CL
   MOV   CL,[BP].PARM2
   MOV   ES:PHYSCPUID,CL

   ;JUMP TO THE KERNEL INITIALIZATION PROGRAM
   JMP KERNEL-INIT

   ;CONTINUE WITH NORMAL ITC SCHEDULER CODE...
```

KERNEL INITIALIZATION SEQUENCE

Figure III-4

into the kernel is the start address of the ITC Scheduler module. All processors will execute the initialization sequence, at the start of the ITC Scheduler, once transfer from the bootstrap program is complete. The start of the initialization sequence is in effect a special entry point into the kernel which is used for initialization only and thus executed only once. All other entries to the ITC Scheduler consist of calls to specific procedures within the module, and therefore, never encounter the initialization sequence.

The first set of instructions in the initialization sequence will allow the parameters passed from the bootstrap program (LOG$CPU$ID and PHYS$CPU$ID) to be popped off the present stack and stored under identical names reserved in the kernel's Processor Data Segment (PRDS) [17]. The PRDS is a per processor data segment that will be utilized by the kernel for specific processor identification. Having completed the transfer of parameters from the bootstrap program, the initialization sequence will then jump to a special initialization program [17] to establish the correct execution environment for the kernel. The initialization program is tasked with initializing the kernel data structures. Specifically the initialization program will cause the idle process to be initialized to running and the kernel loader process will be reflected as ready in the Virtual Processor Map (VPM) [23,17]. Once the proper kernel environment has been established, normal kernel execution can commence. This just requires a transfer of control from the special initialization program to the kernel ITC Scheduler that then schedules the loader process, since it is on the highest priority, ready virtual processor.

2. The Run-time Loader

The Run-time Loader is a kernel process that will be employed to load the application programs from secondary storage. Because the Loader process has a higher priority than the Idle process (the lowest priority- always) and since no other processes are yet defined in the system, the jump to the ITC Scheduler at the end of the bootload phase appears to the kernel as a preempt interrupt of the idle virtual processor. This preempt causes the higher priority Loader process to be scheduled and run on each physical processor.

The kernel Loader process will have the benefit of the operating system primitives provided by the kernel. In particular the ITC Advance and Await [23] procedures will provide for process synchronization and communication during the loading sequence of the application processes.

The details of the Run-time Loader process will be postponed until the next chapter since a significant portion of the mechanism is incorporated in the automatic recovery routine. Once the concepts of system reinitialization have been presented in Chapter IV, the kernel Loader process will be described in detail.

IV. AUTOMATIC RECOVERY DESIGN

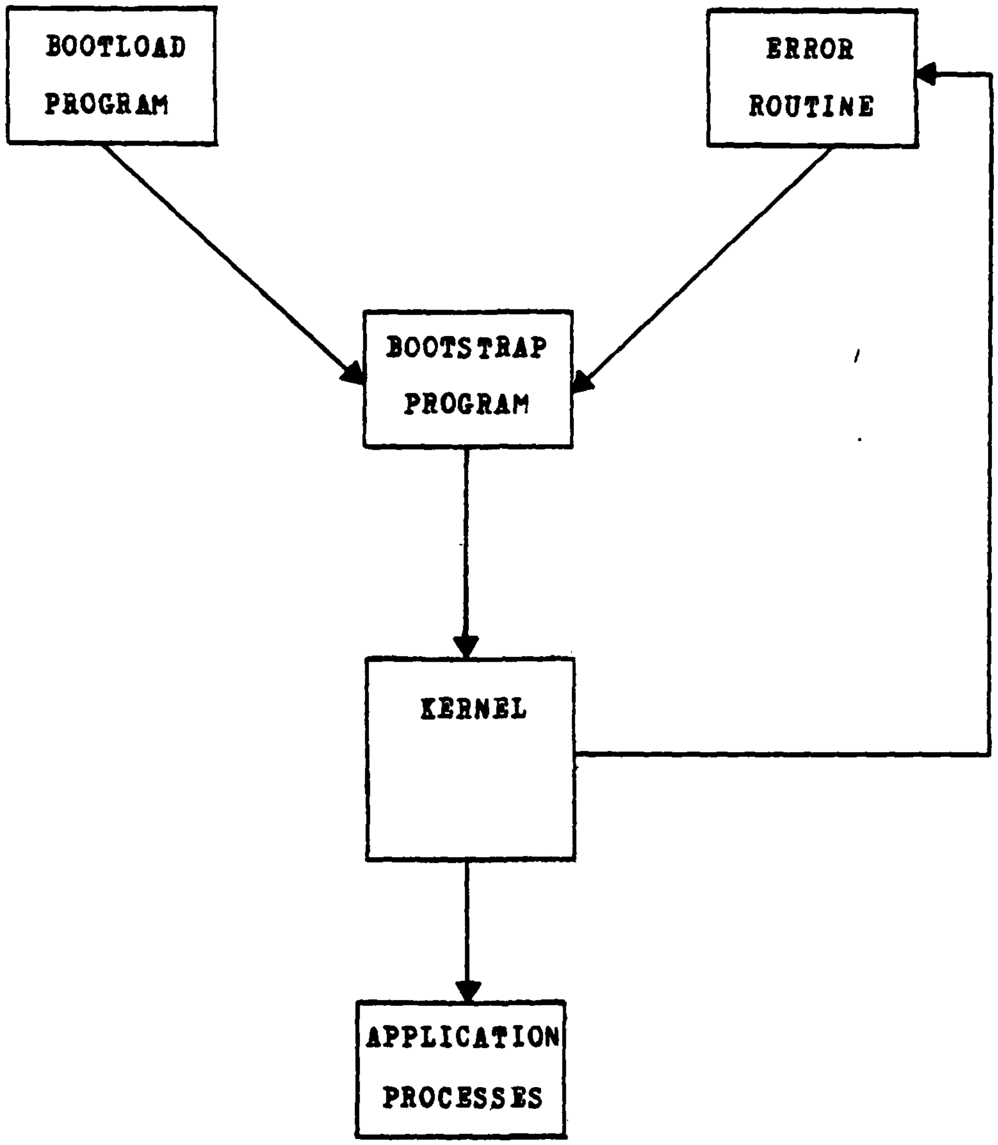
This chapter presents an automatic recovery design that is based on system reinitialization. The mechanism for system initialization, described in the previous chapter, has been modified to form an automatic system recovery routine that integrates with a hierarchical, distributed operating system to support fault-tolerent operation. First a brief overview of the design is presented and then a detailed description of the automatic system recovery mechanism is described.

A. DESIGN OVERVIEW

Automatic recovery begins once a system has detected and diagnosed a component failure. It is the responsibility of an error routine (for the purpose of this discussion encompassing both error detection and diagnosis functions) to indicate the particular component that has generated the system failure. Once the failure has been isolated, by the identification of it's source, it is then the recovery mechanism's responsibility to perform the operations necessary to return the system to a normal, fault-free state.

The automatic recovery technique employed in this design results in a complete reinitialization of the system establishing a predefined initialized state. Upon completion of the automatic recovery routine, the system will have returned to a state identical to that of the original bootstrapped system and will be prepared to begin normal execution. Many of the techniques used for automatic system recovery were previously employed in the initialization routine described in Chapter III. For this reason it is possible to incorporate the automatic recovery mechanism with the initialization routine to provide an overall design that includes both system initialization and automatic system recovery.

System initialization and automatic recovery perform the same basic functions; that of complete system restoration. For initialization the restoration of the system begins from a "cold start" with the activation of a bootload switch, while the automatic recovery process is initiated by an error routine to restore or reinitialize the system. As Figure IV-1 shows, after initialization or automatic recovery has commenced the basic tasks performed are identical. First a bootstrap program is invoked, executing on the bare system hardware, to load the kernel. This is followed by a transfer of control from the bootstrap program to the kernel where an operating system loader routine will be engaged to load the application processes. The distinction between the initialization sequence of events and that of the automatic system recovery routine is based



INITIALIZATION AND RECOVERY SEQUENCE

Figure IV-1

on the fact that initialization is executed only once, establishing the system configuration for the first time, while automatic recovery involves continued reconfiguration and reinitialization for the lifetime of the system.

The contrast between initializing the system for the first time and subsequent reinitialization during automatic recovery is distinguished by the potential loss of system components, due to incorrect performance, during automatic system recovery. Additional tasks must be employed during reinitialization, that are not applicable during initialization, to compensate for the loss of system components. These tasks must specifically deal with system reconfiguration and process relocation in order to return the system to an initialized state that will allow continued normal, fault-free performance.

Complete reinitialization involves reloading, from auxiliary storage, all system processes from the lowest level of the operating system to the user's application programs. The requirement for complete reloading of the system results from the fact that all modules are physically connected by a primary, shared bus (the Multibus [4]) and any faulty component can potentially affect all system modules and data. The automatic recovery mechanism is designed to deal with faulty components on the module level of processor and local memory. Specifially the design calls for the use of the iSBC 86/12A Single Board Microcomputer to be employed as the system component that will be reconfigured during system reintialization.

Elimination of a particular module during automatic system recovery, due to incorrect or faulty performance, will require that the individual processes which were assigned to that module be relocated. The loss of a module as a result of automatic system recovery will require reloading of the system processes on a new hardware configuration, thus tasking the reinitialization routine with memory management during process reloading and relocation.

The real-time recovery tasks developed in this design can be expanded to afford fault-tolerance to a wide spectrum of multiple computer systems. The flexible system environment created through the use of dynamic reconfiguration supports a variety of multi-processor functions. The concepts involved in the automatic recovery mechanism provide the basis for fault-tolerent computing by allowing continued normal system operation after the elimination of faulty components.

B. RECOVERY INTERFACE

Once automatic system recovery commences the fault-tolerence routines involving error detection and diagnosis are assumed to have been completed. As was aleuded to previously, this thesis does not attempt to identify any specific error routines. It is of no consequence to the recovery mechanism how errors were determined, only that they have been diagnosed. Although specific error detection mechanisms are immmaterial to the automatic recovery routine, it is necessary for the interface between the routines to encompass communication and synchronization in order to establish a smooth transistion into the recovery routine. The interface to the recovery mechanism is the responsibility of the error routine and serves the purpose of establishing a predetermined, consistant system state that will always allow automatic system recovery to proceed correctly each time the routine is invoked.

1. <u>The Error Routine</u>

This section briefly outlines the error routine requirements necessary to support automatic system recovery. As was previously mentioned, it is beyond the scope of this thesis to develop the specific error routine mechanism. This section should serve only as a possible example for future development of the error procedure.

The system error routine is required to establish a previously known system state for the interface into the recovery process. This state will simply be defined as the state of the system prior to loading (bootstrapping) the system processes. Additionally the error routine will be required to have performed it's defined task; that of eliminating the faulty module. In this design, that will entail halting the faulty processing module (iSBC 86/12A) so that is can no longer participate in system execution.

The error routine is assumed to be executing on all modules once a fault is detected. An error routine diagnosis program will then determine the faulty module. This could be as the result of a two out of three vote or a test program that indicates the faulty module. In any case the specific faulty module is identified.

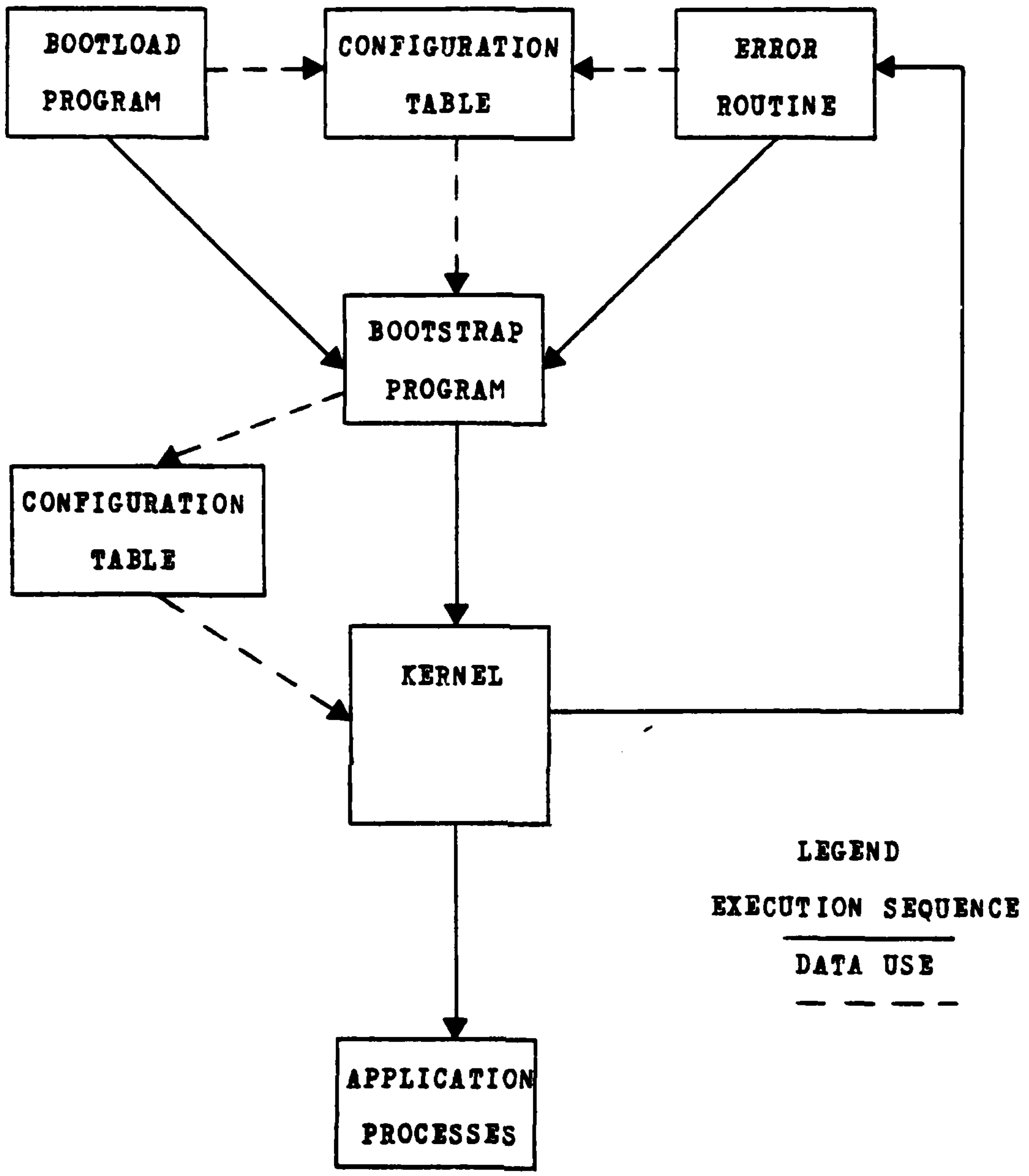
Since the improperly functioning module has been previously determined, the error routine is simply required to halt the faulty processing unit and then initiate the recovery process. The operating system's preempt interrupt provides a relatively straight-forward way for the error routine to eliminate a faulty module. First the error routine will establish the idle process [23] as the highest priority process capable of execution on the faulty processor unit. This is just a matter of altering the priority in the faulty CPU's Virtual Processor Map [23] causing the virtual processor dedicated to the idle process to be the highest priority. Then the particular processor on which the error routine is executing must send a preempt signal to the faulty processor module that will force the faulty module to run the idle process. This will effectively make the improperly performing module unavailable to any other processes. The idle process, running on the faulty module, will then be required to check a system wide error table, indexed by logical CPU number, to determine if a halt should be executed. The error routine will have previously set the halt flag for the faulty processing unit and the result will be the elimination of the failed module from participation in system execution.

Additionally in the event the faulty module has failed completely (i.e., the CPU is unable to execute the idle process), the error routine is tasked with physically disabling the module from the system. This can be accomplished by incorporating in the error routine a hardware "disable" mechanism that will eliminate the faulty module from system interaction.

Once the error routine has eliminated the faulty module from the system it will perform a sequence of tasks that will establish the interface environment for the automatic recovery mechanism. Specifically the error routine will be required to reinitialize the Configuration Table (see Figure IV-2) and then transfer control to the bootstrap program. The Configuration Table is a modified version of the CPU Table designed to support both initialization and reinitialization and will be employed by the bootstrap program in the same manner as described in Chapter III.

a. The Configuration Table

The Configuration Table is a global record structure that will be used primarily to record memory usage and CPU availablity during automatic system recovery. As

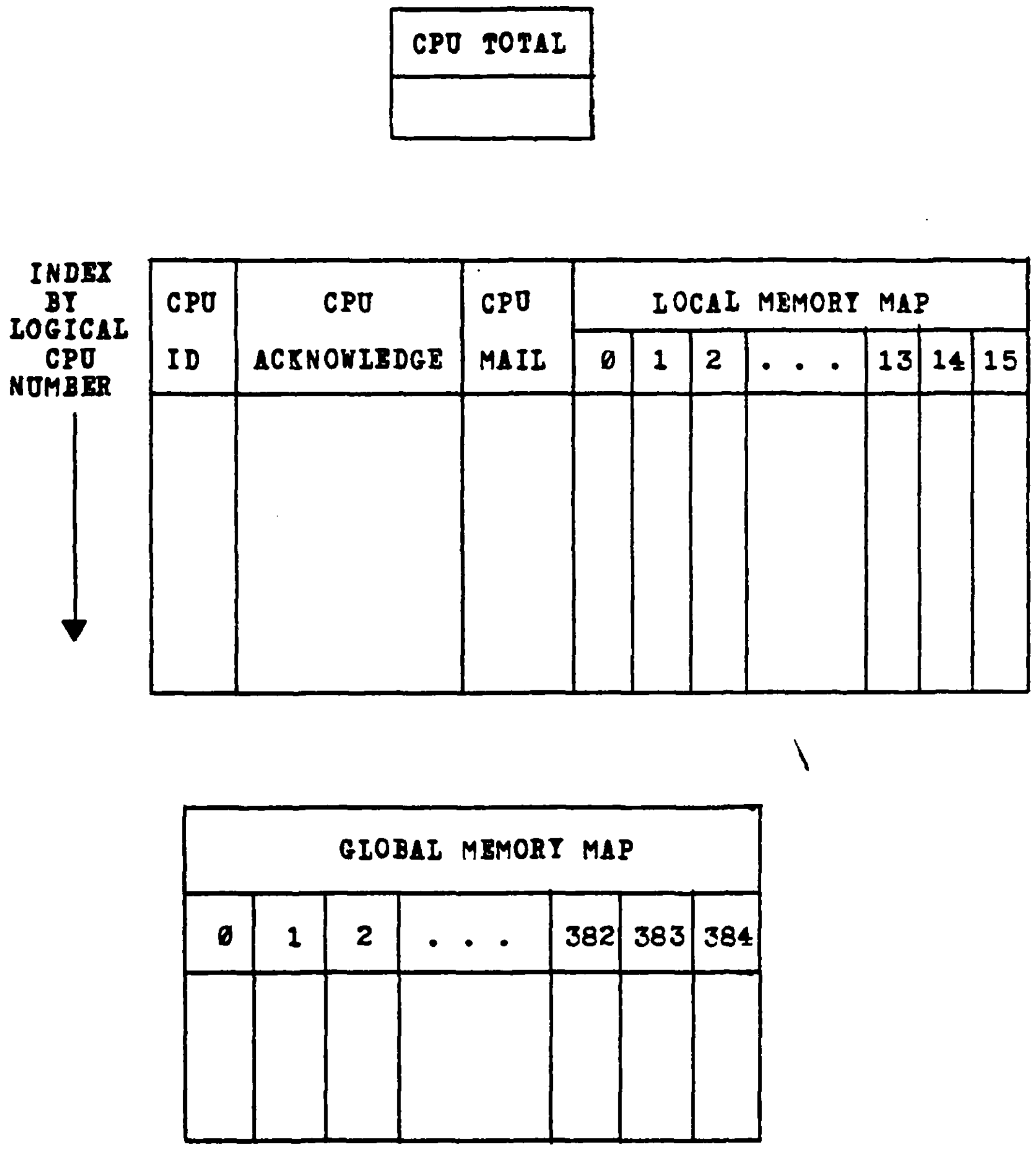


AUTOMATIC RECOVERY SEQUENCE

Figure IV-2

shown in Figure IV-3, three basic structures comprise the Configuration Table. The first, called the CPU Total, will be reinitialized by the error routine to reflect the number of fault-free processors available to the system at the time of automatic recovery. Because the error routine has knowledge of the total processors in the system prior to automatic system recovery, either from the initialization routine or from a previous execution of the automatic recovery process, it can determine the number of properly functioning modules to enter in the CPU Total structure after performing elimination of the faulty module.

The next structure in the Configuration Table is a multiple entry record that is indexed by logical CPU number. The first fields in this structure are identical to the same CPU Table fields described in Chapter III. The error routine will be responsible for reinitializing the unique physical processor serial numbers for each fault-free processor in the system. This essentially involves allowing each processor to access the Configuration Table, one at a time, to enter it's CPU identification number much in the same fashion as the processors were numbered in the bootload program during system initialization. As in the bootload program the logical numbering of the CPUs in the Configuration Table is performed in a random manner.



THE CONFIGURATION TABLE

Figure IV-3

The Configuration Table will also contain a CPU mailbox and a CPU acknowledge entry for each logical processor in the system. These entries will be used during the bootstrap program for CPU synchronization as was the case in the bootstrap program described previosly. Note that the CPU Table used for system initialization in Chapter III is incorporated in the Configuration Table. This allows the system initialization routine to use the Configuration Table structure in the same manner as the CPU Table and provides compatibility between the initialization programs and the automatic recovery routine.

Additionally, the Configuration Table will include a local, per processor memory map and a global memory map that will be used to support the memory allocation mechanism used for reinitialization. To facilitate the recording of memory usage during automatic recovery, memory has been logically subdivided into pages of 256 bytes in length. The global and local memory maps in the Configuration Table are bit maps that will reflect the memory utilization of the system as reloading of the system processes proceeds. Specifically each processor will represent it's 32 kilobytes of local memory using a 16 byte bit map. As shown in Figure IV-3, a 16 byte array is associated with each logical processor number in the Configuration Table structure. Additionally the global memory map, shown in Figure IV-3, will consist of a 384 byte array which will allow the memory allocation mechanism the capability of accounting for the one megabyte of addressable memory minus the possible eight module local memories. Note that although the module memory of each iSBC 86/12A can be divided between local and global memory the real-time system design dedicates all iSBC 86/12A memory (32 kilobytes) to local memory to be used by the 8086 CPU. As a result no global memory will reside on any of the iSBC 86/12As. This means that all global memory will be provided by separate dedicated memory boards.

The Configuration Table is a static structure that is created at system generation time based on the maximum number of modules to be employed in the system and the maximum amount of memory to be utilized. Once the error routine has zeroed all entries in the Configuration Table, then entered the total CPUs available to the system in the CPU Total field and reintialized all the processor's unique ID numbers, it will be required to reload the bootstrap program.

b. The Load CPU

The Load CPU serves as the coordinator of the automatic recovery routine, performing similar duties as that of the Bootload CPU described in Chapter III. The title of Load CPU is assigned to the first CPU to access the Configuration Table during the reinitialization of the unique physical processors serial numbers. The Load CPU is logical CPU number zero in the Configuration Table. Since the reinitialization of the physical processor numbers is accomplished in a random fashion, any one of the fault-free CPUs remaining in the system is capable of being the Load CPU.

The error routine will task the Load CPU with the job of reloading the bootstrap program into global memory. Recall that as in Chapter III, the primary task of the bootstrap program executed during automatic system recovery, is to load the kernel.

2. Recovery Activation

The error routine will activate automatic system recovery by allowing the Load CPU to transfer control from the error program to the bootstrap program it just reloaded into global memory. All remaining processor modules will enter a wait state in their respective error programs. Note that this sequence of events is identical to the action that took place in the bootload program for system initialization. All CPUs, except the Load CPU, will enter an active spinlock in their respective error routines waiting for a signal from the Load CPU in the form of the bootstrap address, before transferring control to the bootstrap program. The error routine wait state is the consistant state all processors (except the Load CPU) will enter during the recovery routine interface and is the state from which system reinitialization will always commence.

The Load CPU will transfer control to the just loaded bootstrap program using an indirect procedure activation (viz., a call by pointer) in the same fashion as the Bootload CPU did in system initialization. The parameters passed to the bootstrap program will include a pointer to the Configuration Table, a pointer to a global bus lock variable that is used to control access to the Configuration Table and the logical processor identification number. Once the Load CPU has transferred control to the bootstrap program and passed the parameters just described, automatic system recovery will commence.

C. OPERATING SYSTEM REINITIALIZATION

Automatic system recovery commences from a predetermined state established during the interface to the automatic recovery routine. The purpose of this defined state is to create a consistant environment from which the reinitialization process can always begin correctly. The previous discussion described the interface state that was determined by the error routine. It is in this state that the first part of reintialization, that of the kernel, begins.

The reinitialization of the kernel is accomplished using a bootstrap program that performs the identical tasks as the bootstrap program described in Chapter III. All processor modules, under the control of the Load CPU, will have the opportunity to execute the global bootstrap program in order to load their respective kernels. Once the Load CPU has transferred control from the error routine to the bootstrap program the actual process of reinitialization will begin.

1. The Bootstrap Program

The primary task of the bootstrap program is to reload the kernel. The first processor to enter the global bootstrap program will be the Load CPU. Recall that all remaining processors are waiting in their respective error routines until the Load CPU signals it is their turn to transfer to the bootstrap program and load their individual kernels.

a. Kernel Reinitialization

The distributed kernel is reinitialized by the bootstrap program which loads each processor module's (iSBC 86/12A) local memory with the required kernel processes. The bootstrap program will perform identically to the bootstrap program described in Chapter III, loading in logical sequence each module's kernel. The details of this portion of kernel reinitialization are related in Chapter III and thus only a brief overview, highlighting the bootstrap program's tasks, will be presented in this section.

The Load CPU, executing in the global bootstrap program, will be tasked to reload each individual module's distributed kernel into a global memory buffer. Once this is accomplished the Load CPU will determine the particular module designated for the kernel just loaded. Using the kernel's designated module identification (affinity) the Load CPU will signal the target processor desired, by filling in the target CPU's mailbox in the Configuration Table with the address of the bootstrap program. After the target processor detects that it's mailbox has been filled, it will exit it's wait state in the error routine program and transfer control to the bootstrap program. The target CPU will then proceed to reload it's kernel file from the global buffer into it's own local memory with the result being a reinitialized kernel. The target processor then signals the Load CPU, via it's acknowledge entry in the Configuration Table, that it has completed reinitializing it's own kernel. The Load CPU will then reload the next kernel from secondary storage in the same fashion. This sequence of events is continued, under control of the Load CPU, until all system modules have had their respective kernels reinitialized.

Upon completion of the kernel reinitialization routine the Load CPU will signal all processor modules by setting it's own acknowledge flag in the Configuration Table. This will force all processors to execute an indirect procedure activation (a call by pointer) to transfer control from the bootstrap program to each modules respective kernel. This jump to the kernel will be accomplished in the same fashion as outlined in Chapter III, only the parameters passed to the kernel in this instance will be of a different variety. In additon to the logical CPU identification of each particular processor performing the control transfer, the arguments will include the location of the Configuration Table (a pointer) and it's global bus lock variable. Note the unique physical processor serial number is not required to be passed as a parameter as it is contained in the Configuration Table.

b. Configuration Table Reinitialization

During the reloading of the distributed kernel each individual CPU has the responsibility of reinitializing the Configuration Table to reflect the memory pages allocated to it's own kernel. Additionally, the Load CPU is tasked with reinitializing the global memory map to identify the memory reserved for the Configuration Table and the global bus lock variable used to control access to the Configuration Table.

Since the bootstrap program executes on the bare system hardware (viz., with no operating system support), as did the bootstrap program of Chapter III, the memory allocation mechanism of the kernel is not available to distribute and record memory usage. This does not present a difficult memory mapping problem, during reinitialization of the kernel, as the programs and data structures loaded by the bootstrap program can all have constant locations in memory. The ability to locate these programs and data structures at absolute addresses is realized by the fact that these processes will be the first reinitialized programs. This means that all the old system code can be over-written.

Each module is responsible for recording, in the Configuration Table, the local memory pages allocated for the kernel it reloads. Since the location and size of the kernel are known, after an individual module has reloaded it's kernel, it is a simple matter to reinitialize the Configuration Table to reflect the memory pages in which the kernel resides.

The Load CPU is responsible for reinitializing the global memory map to reflect the memory allocated to the Configuration Table and it's global bus lock variable. This action is accomplished as the first set of instructions the Load CPU executes in the bootstrap program. The Load CPU first indexes through the global memory map setting the page entries for the Configuration Table and it's bus lock variable to unavailable and all the other page entries to free. Note that the convention used to indicate a free page in the bit map is a one, while zero indicates a page has been allocated. This allows an all zero setting to indicate a full memory map while non-zero entries indicate remaining free pages are available for allocation.

2. Kernel Interface

The transfer of control from the bootstrap program to the kernel, of all system processors available to the system (i.e., not eliminated by the error routine), will proceed in the same fashion as described in Chapter III. The sequence of events executed to interface from the bootstrap program to the kernel will be presented in this section, but the detailed mechanism involved will be left for the reader to review in chapter III.

Recall that the transfer of control to the kernel is executed by all processors after reloading of the kernel (by all modules) is complete. This procedure was required to allow the kernel to commence execution properly with all kernel processes and synchronization structures established in a consistant state.

Once the Load CPU has signalled all CPUs to transfer to their respective kernels the reinitialization of the distributed kernel can be considered complete. The next sequence of events will entail the reinitialization of the application processes. In order to support the relocation routine that will be employed to reload the application proceses the address of the Configuration Table and it's controlling global bus lock variable must be passed to the kernel. Additionally, the logical CPU identification of each processor must be passed to the kernel during individual CPU control transfers. This will ensure the logical identification of each module in the system and facilitate individual processor memory map location during the dynamic relocation process.

The parameters mentioned above are passed to the kernel on the stack of the bootstrap program. The kernel interface sequence of instructions will be required to remove the parameters passed to the kernel on the stack and designate locations in the Processor Data Segment (PRDS) [17] for these structures. Additionally the kernel interface sequence will be required to establish the correct kernel environment for execution by transferring control to a special reinitialization program that will reinitialize the data structures used by the kernel. Recall that the kernel interface sequence of instructions occur in the ITC Scheduler of the operating system [23]. The readers attention is directed to the detailed description of the kernel interface initialization sequence in Chapter III. This procedure performs the identical function as the kernel interface initialization sequence used during automatic system recovery..

D. APPLICATION PROCESS REINITIALIZATION

The reinitialization of the users application processes employs a kernel loader process. It is the responsibility of the kernel loader process to reload the application processes once the distributed kernel has been reinitialized and has restarted execution. Essentially the kernel loader process performs a reinitialization of the application processes, establishing a known correct state (that of the original initialized system) from which the system can restart execution of it's logical tasks.

Reinitialization of the user's application processes begins with each physical processor commencing execution in it's own kernel loader process. The sequence of instructions executed, once the kernel initialization has been completed, to allow the kernel loader process to run are summarized by Wasson [23]. Essentially they entail reinitializing the Virtual Processor Map [23] of every kernel to reflect the loader process as the highest priority process ready to run on any processor. This has been accomplished by the reinitialization of the kernel data structures during reloading. This ensures that all processors will load and run their loader processes first once kernel execution commences.

The reinitialization of application processes involves reloading the application programs using a new system configuration in which faulty modules have been eliminated. Since faulty components are eliminated on the module level of processor and memory (i.e., an iSBC 86/12A) those application processes assigned to a faulty module are reassigned, during reinitialization, to a module that is functioning properly.

The ability to reassign the application processes during reinitialization to different modules (once a module is eliminated) is based on the use of identical modules. Since all processor and local memory units are the same (i.e., all are iSBC 86/12As) the application processes are capable of executing on any module. Note that specific applications programs may impose restrictions that will not allow reassignment to just any available module. These restrictions might be due to the length of a program (i.e., it is too large to be reassigned to a module that already has processes assigned). In this case a spare module might be assigned if available. The specific restrictions imposed by an application process concerning its reassignment will be discussed later in the chapter.

1. Segmentation

The ability of the reinitilization routine to reassign the application processes to different modules during automatic system recovery is dependent on the use of segmented memory. Segmentation allows each application process to have a defined address space that can be specified by a distinct group of segments in memory. Shared segments can exist in the address space of multiple processes for the purpose of inter-process communication, while individual processes can be isolated from other processes by using unique segments that are not shared.

Segmentation of memory is supported by the Intel hardware associated with iSBC 86/12A module. Recall that the one megabyte of addressable memory available to the 8086 CPU provides segments up to 64 kilobytes long [5]. Although explicit segment boundaries are not enforced, the use of a segment manager to allocate memory, based on a predetermined page size and segment length, will allow the manipulation of a processes address space. This, in turn, will support dynamic relocation.

2. Dynamic Relocation

Reassigning the application processes, during reinitialization, is made highly flexible if the ability exists to relocate the segmented address space of the processes. The capability to relocate the application processes facilitates reloading these processes at different locations in a newly assigned module's local memory or in global memory, thus utilizing available memory effectively. The automatic relocation of the application processes, during reinitialization procedure, is known as dynamic relocation.

a. The Compact Compiler Option

Dynamic relocation is made possible if no absolute memory addresses are contained in a processes address space. The ability to dynamically relocate the application processes, during reinitialization, is facilitated by using the compact option of the PL/M-86 compiler [7]. All code compiled using the compact compiler option is placed in either a code, data, stack or optional user defined memory segment depending on its use. Because only these four segments are allowed (i.e., all code is compacted into one of the four segments) the segments remain unchanged during the lifetime of program execution. This means that the Code Segment (CS), Data Segment (DS), and Stack Segment (SS) registers of the 8086 CPU are fixed and thus not changed during program execution. Consequently all code references are reflected as offsets from the CS, DS, or SS registers and no absolute addresses are entered in a processes address space. The placement of offsets in the object code, by the utility locator routine (LOC86) at system generation time, facilitates relocation of a process during reinitialization in that the absolute address of all segments of process can be changed by altering the 8086 CS, DS, or SS registers.

b. The Prologue

All Intel object files, created using the PL/M-86 utility routines [6], invoke a program prologue at the start of execution. This prologue is designed to establish the address space of the program to be executed by setting the appropriate registers in the 8086 CPU. The prologue will differ depending on how the program was compiled. For the automatic system recovery design, the compact compiler option was employed as it provided the most flexible environment for dynamic relocation.

Since all code compiled with the compact option exists in one of four segments [7], the 8086 CPU's CS, DS, and SS registers are required to be set only once as they remain unchanged during program execution. The program prologue of a compact compiled program will set the CS, DS, and SS registers prior to program execution. In order to relocate the application processes, compiled using the compact option, the program prologue for a process must be avoided so that the 8086 CPU registers can be set to reflect a possible new process location after reinitialization. This can be accomplished by creating, essentially, a new program prologue (in the form of an assembly language program, as shown in Figure IV-4) that will not set any of the 8086 CPU registers. The function of this "Start" program for each application process will be simply to perform a short jump to the start of the actual entry point address of the application process. This allows the 8086 CPU registers that define the address space of a process, during execution, to be set to reflect a possible new location of the application process.

The simple start assembly language program will allow the normal program prologue of the application programs to be by-passed (i.e., no CPU registers are set). As Figure IV-4 shows this is accomplished using just the offset of the start address of the application program. This short jump to the application program entry point, using only the address offset, facilitates program relocation by allowing the code to be independent of absolute addresses.

AD-A096 339　NAVAL POSTGRADUATE SCHOOL　MONTEREY CA　F/G 9/2
AUTOMATIC RECOVERY IN A REAL-TIME, DISTRIBUTED MULTIPLE MICROPR--ETC(U)
DEC 80　R L ANDERSON

UNCLASSIFIED　NL

2 OF 2

AD A
096339

```
; START.ASM

; INITIALIZE THE APPLICATION START ADDRESS
; AS A DOUBLE WORD VARIABLE
START-DATA SEGMENT

   APPL-START-ADDR   DD   0000:0006

START-DATA ENDS

START SEGMENT

   ASSUME CS:NOTHING
   ASSUME DS:NOTHING
   ASSUME SS:NOTHING
   ASSUME ES:NOTHING


   ; MOVE THE APPLICATION START ADDRESS
   ; INTO THE AX REGISTER AND DO A SHORT JUMP
   MOV   AX, OFFSET APPL-START-ADDR
   JMP   AX

START ENDS

END
```

START ASSEMBLY LANGUAGE PROGRAM

Figure IV-4

c. The Process Definition Table

The manipulation and relocation of a process' segmented address space, during reinitialization of the application programs, is primarily supported by a global data structure called the Process Definition Table (PDT), as defined by Ross [20]. This structure is created by the system programmer at system generation time and identifies the address space of every application process that will be loaded (or reloaded) to run on the system. Since the address space of every application process is known, prior to commencing system execution (viz., all segment sizes have been established for the run-time, static environment), the PDT entries can be predetermined at system generation time.

The primary function of the PDT is to associate a group of segments with each application process, thus establishing a unique address space for each application process. The PDT is reloaded into global memory at the same time that the reloading of the kernel is accomplished. The kernel loader process then uses the PDT to recreate the application processes as reinitialization is performed.

The PDT, as shown in Figure IV-5, is a static structure, the size of which is predetermined at system generation time as a function of the number of application process to be used in the system. The PDT is indexed by logical process number which will identify the processes to the system reinitialization mechanism. The first entry in

INDEX BY PROCESS NUMBER

PROCESSOR CONFIG. MAPPING					PRIORITY	PROCESS REGISTERS										PROCESS ADDRESS SPACE (PAS)					SEG. R/W MAP
8	7	6	5	4		IP	SP	BP	SI	DI	AX	BX	CX	DX	FL	CS	DS	SS	ES1	. . .	ESN	

THE PROCESS DEFINITION TABLE

Figure IV-5

the PDT, called Processor Configuration Mapping (PCM), is an array that determines the configuration of the system. This array serves to associate, or map, specific logical processors to individual application processes and is indexed, in decreasing order, by the number of modules (iSBC 86/12As) available to the system during the reinitialization routine. The Processor Configuration Mapping entries establish a processor affinity, for a particular application process, as a function of the total processor modules remaining in the system during automatic system recovery.

The ability to dynamically reconfigure the system using the logical CPU affinities designated in the Processor Configuration Mapping is based on the use of identical modules (viz., the unique physical identification of a module is not necessary). For example consider a system which originally consists of eight modules (i.e., eight iSBC 86/12As). The modules are simply assigned to application processes by a logical number between zero and seven in the PCM entry that reflects eight modules are available for system use. Once a module fails, the remaining seven modules are reassigned application processes based on the logical entries in the PCM and the predetermined configuration for seven available processors in the system.

The processor affinities for a particular application process are established at system generation time by the system programmer and must be carefully coordinated to ensure continued system operation as the processors are diminished. Note that a minimum number of processors is usually required to sustain correct system operation and this number is reflected by the last entry of the Processor Configuration Mapping (PCM) array.

Additionally the PDT will contain an entry for the process priority (PRIORITY). This will be used by the kernel to establish a preempt priority during system execution. Following this will be a process register entry (PROC$REG) that can be used to establish any 8086 CPU register settings (other than the segment registers) during the reinitialization of the application processes. In most cases only the Instruction Pointer (IP) will be set and all the other register values will be reinitialized to a null or zero setting.

The last entries in the PDT establish an individual application process' unique address space (PAS). These entries will consist of an array in which the first three entries will be dedicated to the Code Segment (CS), Data Segment (DS) and Stack Segment (SS), respectively, of an application process. The remaining entries will be used, as required, to provide the identification of any external shared segments that exist in a particular application process' address space. The maximum number of external segments are fixed at system generation time and are a function of the application processes and their requirements. The entries in the address space array of the PDT will be unique logical numbers that will identify individual segments in another global data structure, used during reinitialization, called the Global Active Segment Table (GAST). This structure will be described in the next section.

The last field of the Process Definition Table (PDT) is a bit map identifying an individual segment's attributes. In particular this bit map uses a zero (Ø) to signify if a segment is only readable (R) and a one (1) to mark a segment as readable and writable (R/W). A segment attribute will be required by the segment manager in the kernel to determine whether a segment is to be relocated in global or local memory during reinitialization.

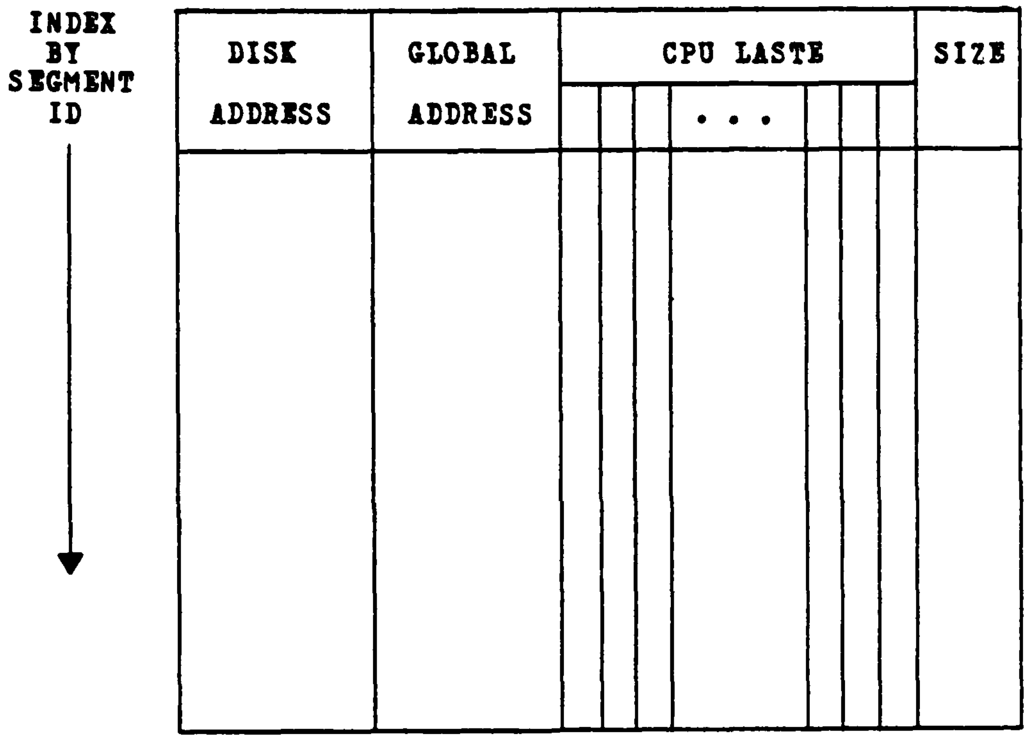
d. The Global Active Segment Table

The Global Active Segment Table (GAST) is a global data base structure employed by the kernel loader process to reinitialize the application processes. It performs essentially the same function as the GAST described by Moore and Gary [14] in their memory manager design; it provides a listing of each individual active segment used in the system (for the run-time, static system design all segments are considered to be active). The GAST identifies the auxiliary storage address of every segment used by the system application processes and associates a logical number, corresponding to the GAST index, with every segment established in memory by the systems programmer.

The GAST, as shown in Figure IV-6, is created, as was the PDT, at system generation time and reload with the kernel. The size of the GAST is determined by the maximum number of application processes in the system and the maximum number of authorized segments per process address space.

The GAST is indexed by segment number. The logical index of each segment in the GAST will be entered in the PDT at system generation time to allow each segment in an application process' address space to be identified. This convention will provide the segment manager process, in the kernel loader, with the ability to access each individual segment in the system for reloading during process initialization.

The secondary disc address of a segment will be contained in the first field of the GAST (DISC$ADDR). This absolute disc address will be used by the kernel loader process to reload the segment during application process reinitialization. A null entry for the disk address indicates that the segment (e.g., a data buffer) must be allocated main storage, but has undefined initial contents. The Global Address field (GLOBAL$ADDR) of the GAST will be used to indicate if a segment resides in global memory. If the global address field is set then the segment is located



THE GLOBAL ACTIVE SEGMENT TABLE

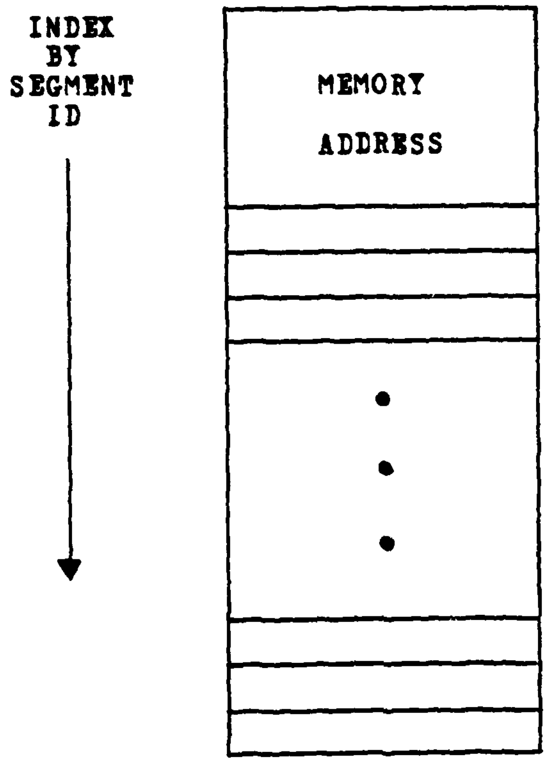
Figure IV-6

in global memory. If the field is null then the segment must be located in local memory.

The CPU Local Active Segment Table Entry (CPU-LASTE) is used as a connected processor list. The field is an array structure which is as large as the maximum number of processors originally allocated for the system. The entries in this field provide an index into each processor's Local Active Segment Table (LAST) and will be used by the segment manager in the kernel loader process to manipulate segments during process reinitialization. The length of segment is contained in the Size Field (SIZE) of the GAST. This entry is used by the segment manager process of the kernel loader to allocate the appropriate amount of memory for the segment during the reloading of application process reinitialization.

e. The Local Active Segment Table

The Local Active Segment Table (LAST) is employed during reinitialization for the purpose of memory allocation in the same fashion that Moore and Gary [14] used it in their Memory Management Unit. The LAST (see Figure IV-7) is a processor-local data base in the form of an array that records the local memory location of all segments reloaded on a particular processor module. The index into the LAST is reflected in the GAST's connected processor list (CPU LASTE) for each individual segment in the system. The LAST entry in the GAST is used by the kernel segment



THE LOCAL ACTIVE SEGMENT TABLE

Figure IV-7

manager routine to locate segments previously reloaded that must be moved to global memory due to their being shared and writable.

3. The Kernel Loader Process

Reinitialization of the application processes begins once all processor modules have entered the kernel Loader process (see Appendix D). Recall that the kernel has been reinitialized so that once it starts execution, the Loader process, being the highest priority process ready to run, will be the first kernel process executed. Since the logical processor number of every CPU was passed, when control was transferred from the bootstrap program to the kernel, all modules maintain their logical identity. This means that one particular CPU still has the title of Load CPU. It is this processor unit that will coordinate application process reinitialization during automatic system recovery.

The Kernel Loader process is required to reload the application processes sequentially according to their entry in the Process Definition Table. Reloading of the individual applications processes one at a time (viz., not simultaneously) is necessary primarily due to hardware limitations. In particular, as described in Chapter III, not all processors will have access to secondary storage thus requiring the Load CPU to perform system I/O using a primary memory global buffer that the remaining CPUs can access.

a. The Load CPU

The Load CPU will execute some instructions in the kernel Loader process that the other processors will not. In particular the Load CPU will have the responsibility of sequentially indexing through the Process Definition Table (PDT) identifying each application process and the physical module into which it will be reloaded. The association of a processor and an application process to be reloaded is accomplished using the Processor Configuration Mapping field (PCM) of the PDT. Recall that this mapping is based on the number of physical CPU's available to the system at the time of reintialization. The mapping configuration of the processors includes all combinations of processors from the maximum available down to the minimum required to continue correct system execution. The Load CPU will not do the actual reloading of the application process, but will signal (via the ITC Advance procedure [23]) the processor module associated with the process, in the PDT, to perform the task. Note that although the automatic recovery mechanism is based on the use of identical processor modules, future expansion of the design might include special processors (i.e.,a Multiply CPU). It would then be necessary to use the Configuration Table to identify a specific physical processor and it's associated logical number.

The particular processor signalled by the Load

CPU is a function of the mapping configuration associated with an applications process in the PDT and the number of CPUs available to the system during reintialization. Note that if the processor required to reload the application process is the Load CPU, the reinitialization of that particular process is performed by the Load CPU. After accomplishing the reloading, the Load CPU will just index to the next process in the PDT.

Once the Load CPU has determined the CPU affinity (the processor associated with a process through the configuration mapping) for a particular process, and signalled (via ITC Advance) the target modules loader process, the Load CPU will enter a wait state (The reintialization of the application processes uses the ITC eventcount synchronization procedures of Advance and Await [23]). The Load CPU will remain in a wait state until the target processor signals (by an advance on the Load CPU's eventcount) it has reloaded, and thus reinitialized, the assigned application process. This sequence of events is repeated until all applications processes listed in the PDT are loaded into the modules they have been assigned to.

While the Load CPU is indexing through the PDT, signalling the appropriate CPUs when it is their turn to reinitialize a particular application process, the remaining processors will have entered a wait state in their respective kernel loader processes. This synchronization is similar to that performed in Chapter III, only the more flexible kernel eventcount primitives are now available to support processor communication. Once a processor, other than the Load CPU, has completed the reinitialization process, it will return to a wait state, remaining in that state until signalled to reinitialize another application process or until system restart is executed.

b. Swap-in

The Swap-in procedure is called by the kernel loader process to reload, from secondary storage, an application process. Swap-in is designed to reload a specific segment in the address space of a process and return the start address of that relocated segment. Moore and Gary [14] originally developed the Swap-in routine for their memory management unit and it is a modified version of their design that is used in the Kernel Loader Process.

The ability to incorporate a portion of the Memory Management Unit designed by Moore and Gary is the result of the fact that the Memory Management Unit design and the Automatic System Recovery mechanism are based on the same family of distributed operating systems originally developed by O'Connell and Richardson [15]. The hierarchal design of the operating system provides a significant advantage in that it is relatively hardware independent and thus compatibility between systems is feasible.

When signalled (by an eventcount advance) to reload an application process, the target CPU will be required to sequentially index through the address space of that process in the PDT. Swap-in will be repeatedly called, by the target processor's Kernel Loader, to reload each individual segment in the process' address space. Each time Swap-in is called it is passed the logical segment number in the PAS array of the PDT. Recall that the logical segment number is used to index into the GAST. Swap-in will be required to use the logical segment number index, in the GAST, to determine the segments absolute disc address on an auxiliary storage device (i.e., a hard disc).

Once Swap-in has established a secondary storage address, it will move the targeted segment into primary memory. The procedure for determing if local or global memory should be allocated is defined by Moore and Gary [14]. In particular three conditions can be encountered during the invocation of Swap-in. The segment can already be located in global memory, the segment can be located in one or more local memories or the segment may not have been previously reloaded during this activation of the automatic recovery routine.

If the segment has not been previously reloaded (i.e.,the GAST Global Address and the CPU LASTE fields are null) then the segment is reloaded in local memory as defined by the process affinity and the appropriate entries in the GAST's connected processor list (CPU LASTE) and the

LAST are made. If the segment has been previously reloaded into global memory (as evidence of the GAST reflecting a global address) then it is not necessary to reload the segment. Only the GAST and the LAST need to be updated. Finally if the segment already resides in one or more local memories, it must be determined if the segment is writable. This is accomplished using the PDT Read/Write bit map. If the segment is writable and located in another modules local memory (as reflected by the GAST's connected processor list; CPU LASTE) it must be moved to global memory where it can be shared and the global address in the GAST filled in. If the segment is only readable then is may be allocated local memory and the LAST updated.

Once the memory space has been allocated for the segment, as determined by the size field in the GAST, Swap-in will reload the segment and update the Configuration Table memory maps; returning the segment location to the kernel loader process. The loader process will then enter the segment's location in the Process Parameter Block (PPB). The PPB is a local data structure that is used to record all the locations of the segments in the process' address space reloaded by Swap-in.

The sequence of events executed, once Swap-in is called, will be repeated until the Loader Process has indexed completely through the PAS array or until a null entry is discovered in the PAS indicating all the process segments have been reloaded. The Loader Process will then call Create-process, passing the locations of the segments just loaded, to complete the reintialization process.

c. Create-process

The Kernel Loader process will call the procedure Create-process to culminate the reinitialization of the application processes. The Create-process routine is an operating system (kernel) routine designed by Wasson [23] and implemented by Rapanzikos [17]. Essentially it reinitializes entries in the process' stack segment that define the process' address space. The process' stack is then used by the kernel to establish a particular application process' run-time environment.

Create-process will be passed the address of the Process Parameter Block (PPB) each time it is activated by a particular CPU Loader process. Recall that the PPB is a local data base used to record the locations of all segments in the application process' address space. The Stack Segment (SS) for each application process will be created using the PPB and the PDT processor register array (PROC$REG). Once Create-process has reestablished a process' address space and reinitialized the register values on the application process' stack it will place the process in a wait state. All processes are recreated in a wait state by Create-process waiting for a system start event (i.e., an Advance on the system start eventcount [17]). Control will then return to the kernel Loader process.

E. RESTART

Once the Load CPU has indexed completely through the PDT the task of application process reintialization is complete. The Load CPU is then required to restart the system so that normal, fault-free execution can resume. This is accomplished by the Load CPU performing an Advance [17] on the system start eventcount. Recall that all application processes are recreated by Create-process suspended in a wait state waiting for the system start eventcount to be advanced. After this event takes place all processors will resume normal operation by executing the highest priority application process assigned.

F. APPLICATION PROCESS STRUCTURE

In order to facilitate dynamic relocation during the automatic system recovery process, some restrictions must be imposed on the structure of the applications programs. It is the purpose of this section to outline these restrictions and additionally provide some insight into their requirement in order that the applications programmer might better perform his programming tasks.

Each application process is determined by a segmented address space that can be defined by unique code, data, and stack segments (using the compact compiler option [7]). Since these segments are unique (viz., not shared) a scheme for segment sharing, to facilitate inter-process communication and synchronization is required.

Shared segments are created, at system generation time, by adding additional segments to a process' address space. These external segments are then reflected in the PDT, associated with each particular application process, depending on process communication and synchronization requirements. The external segments of each process will be reloaded during process reinitialization and as a result of the procedure Create-process, their locations will be placed in the unique stack segment of each individual application process. The stack of each process is, in effect, a unique description segment that contains pointers to all segments in a particular application process' address space. Hardware segmentation then allows the stack segment of an application process to be employed as a parameter list of pointers as described below.

When system automatic recovery occurs, all application processes are recreated by the reintialization routine and thus the external shared segments, as well as the unique code, local data and stack segments, are updated to reflect any changes in segment location. This results in a newly created stack segment that will reflect the reinitialized address space of an application process.

1. <u>The Entry Point</u>

The restriction placed on the structure of an application process is directed at the entry point or start address of the intial procedure. When the kernel activiates a particular application process it will use the stack segment of the process to set the code and data segment registers of the 8086 CPU. Since there are not enough physical registers to allow all external segments in a process to be set, a scheme must be devised so that the process can reference all it's external segments.

The convention to do this exploits the entry point to the application process. This will take the form of a procedure in which the external segment locations will be passed as pointers. Requiring the application process start address to be a procedure entrance will permit the process to use the preset external system pointers on the process' stack to define the formal procedure parameters of the application program. Note that the stack pointer (SP) is set (as defined at system generation time) to indicate the first external segment pointer on the stack.

The applications programmer need only be concerned with parameter ordering in the applications process. The burden of parameter organization, in terms of stack structure, rests with the system programmer at system generation time. Specifically the systems programmer is required to make the appropriate entries in the Process

Definition Table (PDT) to provide the logical ordering of the external pointers in the formal parameter list of the application procedure.

2. External Variables

The external segment pointers, contained in the formal parameter list of the application procedures are declared as PL/M-86 pointer variables. The applications programmer is then required to use these pointer variables to reference PL/M-86 based variables [5]. This action will result in the process' external segment base addresses being used as pointers for addressing the external shared data structures employed in the application process for inter-process communication and synchronization.

V. CONCLUSIONS

A. SUMMARY OF RESULTS

This thesis has focused on a technique for automatic system recovery designed to provide the fault-tolerant operation of a real-time, distributed multiple microcomputer system. The initialization mechanism developed by Ross [20] was implemented and tested as the first phase of the thesis effort and proved to be a solid base from which reinitialization could be accomplished. To support the reinitialization routine, which employed complete reloading of the system processes, a method of dynamic relocation exploiting the Intel hardware was developed. This lead to the ability of the system to dynamically reconfigure after the elimination of a faulty system module.

The fundamental concepts developed as the result of the research efforts of this thesis provide the basis for fault-tolerance in a system where temporary data loss is a tolerable condition. The ability to completely reinitialize the system while eliminating faulty components is a desirable attribute in many real-time systems. The automatic system recovery design presented in this thesis is the basis for fault-tolerance in a real-time system that has a multiple microprocessor environment.

B. FOLLOW-ON WORK

This thesis addressed only one aspect of fault-tolerance; that of fault recovery. As the introduction revealed, the elements of fault-detection and fault-diagnosis are usually included in a fault-tolerant computer design. Research concerning fault detection and fault diagnosis will provide a challenging area for follow-on work. Specifically the error routine discussed in Chapter IV must be developed to support the automatic system recovery mechanism. Only with fault detection and diagnosis routines incorporated will the automatic recovery routine provide complete fault tolerance for the multiple microcomputer system.

Dynamic reconfiguration in the automatic system recovery design revolves around the processor/memory module (the iSBC 86/12A). Further research might specifically investigate the separate reinitialization of only faulty memory. The logical extension of the recovery mechanism lends itself to the possibility of saving the fault-free portions of memory in the form of the PDT and GAST. These data bases would then allow the error routine to eliminate specific sections of faulty memory and record the memory removed. This, in turn, would allow a reduced reloading requirement and thus a more expeditious execution of the automatic system recovery routine.

The automatic recovery design presented by this thesis provides a basis for fault recovery. Further development of the design could proceed in numerous directions with the concepts of dynamic relocation and reconfiguration facilitating a variety of specialized designs. For example, an expansion of the automatic recovery mechanism might include check-pointing, where data processed prior to a system failure could be saved; thus reducing the reintialization requirements. The automatic recovery mechanism might also be used in conjunction with other recovery techniques. In particular reinitialization might be used in a system that employs redundancy. A specific group (i.e., cluster) of faulty microcomputers could be reinitialized to eliminate the faulty module while a parallel cluster is substituted to perform the identical computations.

The automatic system recovery mechanism was developed to integrate with a distributed hierarchical operating system. The original distributed operating system kernel implementation developed by Wasson [23] was not specifically designed to incorporate fault-tolerance. Although this thesis attempted to provide the interface to the operating system the continued development of the kernel will necessitate additional follow-on work to ensure a compatible integration of the automatic system recovery mechanism with the kernel.

APPENDIX A. SYSTEM INITIALIZATION IMPLEMENTATION

A. OBJECTIVES

This appendix is provided to further acquaint the reader with the system initialization mechanism presented in this thesis. To demonstrate the initialization capability provided by the program listings in Appendix B and C, a test program was developed to simultate an operating system kernel. (The test program was required as the previous kernel implementation was not specifically designed to interface with the recovery mechanism). The simulated kernel was then loaded by multiple iSBC 86/12A single board computers in the same fashion as described in Chapter III, using the same hardware support outlined in Chapter II.

B. THE SIMULATED KERNEL

The simulated kernel program in Figure A-1 was loaded by all iSBC 86/12As and was used to demonstrate the ability of the initialization mechanism to transfer control to the kernel and then commence system execution. The demonstration called for each iSBC 86/12A to have a CRT connected to it's serial I/O port. Once all simulated kernels were loaded and execution transferred to each particular iSBC 86/12A kernel, the simulated kernel caused the logical CPU number and the unique physical CPU ID of each processor module (iSBC

86/12A) to be displayed on their respective CRTs.

C. DEMONSTRATION ENVIRONMENT

The demonstration environment for loading the simulated kernel included all the hardware support described in Chapter II, but due to limited resources only a maximum of three iSBC 86/12As were used instead of the eight planned for. This required two bootload programs similar to the listing in Figure B-2 (only the unique physical IDs will differ) and a bootload program (used for the MDS-connected iSBC 86/12A and thus the bootload CPU) identical to the listing in Figure B-1.

D. SYSTEM ACTIVATION

For demonstration the bootload programs were placed in RAM, as described in Chapter III. To initially load all three iSBC 86/12A boards with their respective bootload programs the iSBC 957A-iSBC 86/12A interface and execution package was employed. In particular the monitor command LOAD was executed to load an individual bootstrap program into the MDS-connected iSBC 86/12A's local memory. Once this was accomplished the monitor MOVE command was used to move the bootstrap program to the appropriate iSBC 86/12A. (Note that since the local memory of one iSBC 86/12A cannot be addressed by another iSBC 86/12A the equivalent global address of a particular iSBC 86/12A local memory was used to move the code. Also the MOVE command does not alter any code to reflect a new location; it only provides an explicit transfer of code). Additionally the monitor MOVE command was employed to move the four bytes of the bootload interrupt vector to the designated iSBC 86/12A, again using the global address.

The process of loading an individual bootload program and it's interrupt vector into local memory of the MDS-connected iSBC 86/12A and then moving that code to the identical spot in the targeted iSBC 86/12A (using its global memory for that location) was repeated for both iSBC 86/12A's not connected to the MDS. Finally the bootload program for the MDS-connected iSBC 86/12A was loaded and the initialization mechanism was activated, using the simulated bootload switch: *the INTR button on the iCS-80* chassis. Note that it was necesary to start the MDS-connected iSBC 86/12A executing a loop, as the MDS interfered with the non-maskable interrupt, but that all other iSBC 86/12As commenced execution of the initialization routine from their respective monitors.

```
                                                  PAGE   1

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE KERNELHEXMOD
OBJECT MODULE PLACED IN :F1:KERNEL.OBJ
COMPILER INVOKED BY:  PLM86 :F1:KERNEL.SRC


                /**********     KERNEL.SRC     FILE     20 NOV 80     ***********/

                /* BEGIN KERNEL HEX FILE MODULE */
  1             KERNEL$HEX$MOD:  DO;

  2    1           DECLARE I BYTE;

  3    1           DECLARE K1MSG(*) BYTE INITIAL('LOGICAL CPU ID = ');
  4    1           DECLARE K2MSG(*) BYTE INITIAL('PHYSICAL CPU ID = ');
  5    1           DECLARE KMSG(*) BYTE INITIAL('ENTERED KERNEL ');

                   /* THIS PROCEDURE OUTPUTS CHARACTERS TO THE CRT */
  6    1           OUT$CHAR: PROCEDURE(CHAR);
  7    2              DECLARE CHAR BYTE;
  8    2              DO WHILE(INPUT(0DAH) AND 01H) = 0; END;
 10    2              OUTPUT(0D8H) = CHAR;
 11    2           END OUT$CHAR;

                   /* THIS PROCEDURE OUTPUTS HEX NUMBERS TO THE CRT */
 12    1          OUT$HEX: PROCEDURE(B);
 13    2              DECLARE B BYTE;
 14    2              DECLARE ASCII(*) BYTE DATA ('0123456789ABCDEF');
 15    2              CALL OUT$CHAR(ASCII(SHR(B,4) AND 0FH));
 16    2              CALL OUT$CHAR(ASCII(B AND 0FH));
 17    2           END OUT$HEX;
```

SIMULATED KERNEL LISTING

Figure A-1

```
PL/M-86 COMPILER    KERNELHEXMOD

                  $EJECT
                        /* THIS ISBC 957A PROCEDURE IS USED DURING THE DEBUGGING
                           AND DEVELOPMENT PHASE TO RETURN TO THE ISBC 86/12A
                           MONITOR */
  18    1               EXIT: PROCEDURE EXTERNAL;
  19    2               END EXIT;

                  /*****************************************************************/
                  /*     KERN$INFACE                                               */
                  /*-------------------------------------------------------------*/
                  /*     THIS IS A TEST ROUTINE USED TO SIMULATE THE ENTRY INTO    */
                  /* THE KERNEL. IT OUTPUTS THE LOGICAL CPU NUMBER AND THE UNIQUE  */
                  /* PHYSICAL CPU SERIAL NUMBER TO THE CRT AND THEN RETURNS TO THE */
                  /* 86/12A MONITOR.                                               */
                  /*****************************************************************/

  20    1            KERN$INFACE: PROCEDURE(LOG$CPU$ID,PHYS$CPU$ID);

  21    2               DECLARE LOG$CPU$ID BYTE;
  22    2               DECLARE PHYS$CPU$ID BYTE;
  23    2               DECLARE CR LITERALLY '0DH',
                                LF LITERALLY '0AH';

  24    2               CALL OUT$CHAR(CR);
  25    2               CALL OUT$CHAR(LF);
  26    2            DO I = 0 TO 14;
  27    3               CALL OUT$CHAR(KMSG(I));
  28    3            END;
  29    2               CALL OUT$CHAR(CR);
  30    2               CALL OUT$CHAR(LF);
  31    2               DO I = 0 TO 16;
  32    3                  CALL OUT$CHAR(K1MSG(I));
  33    3               END;
```

SIMULATED KERNEL LISTING

Figure A-1 (cont'd)

```
PL/M-86 COMPILER    KERNELHEXMOD


                 $EJECT
  34   2                   CALL OUT$HEX(LOG$CPU$ID);
  35   2                   CALL OUT$CHAR(CR);
  36   2                   CALL OUT$CHAR(LF);
  37   2                   DO I = 0 TO 17;
  38   3                      CALL OUT$CHAR(K2MSG(I));
  39   3                   END;
  40   2                   CALL OUT$HEX(PHYS$CPU$ID);
  41   2                   CALL EXIT;

  42   2                   END KERN$INFACE;  /* END PROCEDURE */

  43   1           END;    /*  KERNAL$HEX$MOD  */



MODULE INFORMATION:

     CODE AREA SIZE     = 00EBH    235D
     CONSTANT AREA SIZE = 0010H     16D
     VARIABLE AREA SIZE = 0033H     51D
     MAXIMUM STACK SIZE = 0012H     18D
     71 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

SIMULATED KERNEL LISTING

Figure A-1 (cont'd)

APPENDIX B. BOOTLOAD PROGRAM LISTING

```
PL/M-86 COMPILER    INITBLCPUMOD

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE INITBLCPUMOD
OBJECT MODULE PLACED IN :F1:INBOOT.OBJ
COMPILER INVOKED BY:  PLM86 :F1:INBOOT.SRC LARGE


              /**********     INBOOT.SRC     FILE     27 ØCT 8Ø     ***********/

              /* BEGIN INITIALIZE BOOTLOAD CPU MODULE */
 1            INIT$BLCPU$MOD:  DO;

              /*****************************************************************/
              /*     LOCAL DATA DECLARTIONS                                    */
              /*****************************************************************/

 2    1         DECLARE LOG$CPU$ID BYTE,
                        ST$BTSTRP$ADR POINTER,
                        STAT$BTSTRP WORD,
                        PHYS$CPU$ID    LITERALLY    '13',
                        BOOTLOAD$CPU   LITERALLY    'Ø';

              /*****************************************************************/
              /*     EXTERNAL GLOBAL DATA DECLARATIONS                         */
              /*****************************************************************/

 3    1         DECLARE CPU$TBL$LOCK BYTE EXTERNAL,
                        LOG$CPU$NUM  BYTE EXTERNAL;

 4    1         DECLARE CPU$TABLE(8) STRUCTURE
                        (CPU$ID BYTE, CPU$ACK BYTE,
                         CPU$MAIL POINTER, CPU$TOTAL BYTE) EXTERNAL;
```

MDS CONNECTED BOOTLOAD PROGRAM

Figure B-1

```
PL/M-86 COMPILER    INITBLCPUMOD

                 $EJECT
                 /*****************************************************************/
                 /*      EXTERNAL ISBC 957A I/O SYSTEM PROCEDURES                 */
                 /*****************************************************************/

  5   1             LOAD: PROCEDURE(FILENAME,BIAS,SWITCH,ENTRY,STATUS) EXTERNAL;
  6   2                DECLARE (FILENAME,ENTRY,STATUS) POINTER;
  7   2                DECLARE (BIAS,SWITCH) WORD;
  8   2             END LOAD;

  9   1             EXIT: PROCEDURE EXTERNAL;
 10   2             END EXIT;

                 /*****************************************************************/
                 /*      LOCAL PROCEDURES                                         */
                 /*****************************************************************/

                 /*****************************************************************/
                 /*      CPU$WAIT                                                 */
                 /*---------------------------------------------------------------*/
                 /*      CAUSES THE NON-BOOTLOAD CPU'S TO WAIT, IN A SPIN LOCK,   */
                 /*      UNTIL THE BOOTLOAD CPU SENDS THE ADDRESS OF THE BOOT-    */
                 /*      STRAP PROGRAM INDICATING THAT THIS PARTICULAR CPU CAN    */
                 /*      CONTINUE AND EXECUTE THE BOOTSTRAP PROGRAM.              */
                 /*****************************************************************/

 11   1             CPU$WAIT: PROCEDURE(WAIT$CPU$ID);

 12   2                DECLARE WAIT$CPU$ID BYTE,
                               BTSTRP$FLAG BYTE,
                               READY       LITERALLY '01',
                               NOT$READY   LITERALLY '00',
                               NULL         LITERALLY '00';
```

MDS CONNECTED BOOTLOAD PROGRAM

Figure B-1 (cont'd)

```
                  $EJECT
                        /* INITIALIZE LOCK TO SPIN */
 13   2                 BTSTRP$FLAG = NOT$READY;
 14   2                 DO WHILE BTSTRP$FLAG <> READY;
 15   3                    DO WHILE LOCKSET(@CPU$TBL$LOCK,1);
 16   4                    END;
                           /* CHECK TO SEE IF CPU MAIL BOX HAS BEEN SET FROM NULL TO
                              BOOTSTRAP PROGRAM ADDRESS BY BOOTLOAD CPU */
 17   3                    IF CPU$TABLE(LOG$CPU$ID).CPU$MAIL <> NULL THEN
                              /* IF BOOTSTRAP ADDRESS IS IN MAIL BOX THEN SET READY
                                 AND EXIT SPIN LOCK */
 18   3                       BTSTRP$FLAG = READY;
 19   3                    CPU$TBL$LOCK = 0;
 20   3                    END;     /* DO WHILE */
 21   2                 RETURN;     /* END CPU$WAIT PROCEDURE */
 22   2                 END CPU$WAIT;

                  /*****************************************************************/
                  /*    BOOTLOAD$INTR                                              */
                  /*---------------------------------------------------------------*/
                  /*    ACTIVATING 'INTR' BUTTON ON 86/12 CHASSIS CAUSES JUMP      */
                  /*    TO THIS PROCEDURE. THE CPU$TABLE IS THEN ACCESSED INORDER*/
                  /*    TO PROVIDE A LOGICAL AND PHYSICAL ID FOR THIS CPU. IF      */
                  /*    THIS CPU BECOMES THE 'BOOTLOAD' CPU (LOGICALLY = 0) THEN */
                  /*    IT LOADS THE BOOTSTRAP PROGRAM AND JUMPS TO IT. OTHER-    */
                  /*    WISE IF ENTERS A WAIT STATE.                               */
                  /*****************************************************************/

 23   1              BOOTLOAD$INTR: PROCEDURE INTERRUPT 02;

 24   2                 DECLARE I BYTE;

 25   2                 DO WHILE LOCKSET(@CPU$TBL$LOCK,1);
 26   3                 END;
```

MDS CONNECTED BOOTLOAD PROGRAM

Figure B-1 (cont'd)

```
                   $EJECT
                         /* SET UNIQUE PHYSICAL CPU ID IN CPU$TABLE */
 27   2                  CPU$TABLE(LOG$CPU$NUM).CPU$ID = PHYS$CPU$ID;
                         /* INCREMENT BOOTLOAD CPU COUNT */
 28   2                  CPU$TABLE(BOOTLOAD$CPU).CPU$TOTAL = LOG$CPU$NUM + 1;
                         /* SET LOGICAL CPU ID */
 29   2                  LOG$CPU$ID = LOG$CPU$NUM;
                         /* INCREMENT THE CPU NUMBER FOR THE NEXT CPU */
 30   2                  LOG$CPU$NUM = LOG$CPU$NUM + 1;
 31   2                  CPU$TBL$LOCK = 0;
                         /* IF CPU IS THE BOOTLOAD CPU */
 32   2                  IF LOG$CPU$ID = 0 THEN
 33   2                     DO;
                           /* CREATE TIME DELAY TO ALLOW OTHER CPU'S TIME TO ACCESS
                              CPU$TABLE AND IDENTIFY THEMSELVES */
 34   3                    DO I = 1 TO 10;
 35   4                       CALL TIME(100);
 36   4                    END;
                           /* LOAD BOOTSTRAP PROGRAM INTO GLOBAL MEMORY */
 37   3                    CALL LOAD(@(':F1:BTSTRP '),0,0,@ST$BTSTRP$ADR,@STAT$BTSTRP);
 38   3                  END;     /* IF */

                         /* IF NOT BOOTLOAD CPU (LOG$CPU$ID <> 0) */
                         ELSE
 39   2                     DO;
                           /* ENTER WAIT STATE BY EXECUTING SPIN LOCK */
 40   3                    CALL CPU$WAIT(LOG$CPU$ID);
                           /* SET BOOTSTRAP PROGRAM ADDRESS, THAT IS PASSED TO THE
                              CPU MAILBOX FROM THE BOOTLOAD CPU, SIGNALLING TIME
                              TO JUMP TO THE BOOTSTRAP PROGRAM */
 41   3                    ST$BTSTRP$ADR = CPU$TABLE(LOG$CPU$ID).CPU$MAIL;
 42   3                    END;     /* ELSE */
```

MDS CONNECTED BOOTLOAD PROGRAM

Figure B-1 (cont'd)

```
PL/M-86 COMPILER    INITBLCPUMOD


                  $EJECT
                        /* JUMP TO BOOTSTRAP PROGRAM IN GLOBAL MEMORY */
  43   2                CALL ST$BTSTRP$ADR(LOG$CPU$ID,@CPU$TABLE,@CPU$TBL$LOCK);

  44   2             END BOOTLOAD$INTR;     /* END INTERRUPT PROCEDURE */

                     /* MAIN PROGRAM - CREATES INFINITE EXECUTION LOOP ONLY IN
                        THE BOOTLOAD CPU CONNECTED TO THE MDS. */

  45   1             DO WHILE 01;
  46   2             END;

  47   1          END;     /* INIT$BLCPU$MOD */


MODULE INFORMATION:

     CODE AREA SIZE     = 0197H    407D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0009H      9D
     MAXIMUM STACK SIZE = 0034H     52D
     137 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

MDS CONNECTED BOOTLOAD PROGRAM

Figure B-1 (cont'd)

```
PL/M-86 COMPILER    INITCPU1MOD

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE INITCPU1MOD
OBJECT MODULE PLACED IN :F1:INCPU1.OBJ
COMPILER INVOKED BY:  PLM86 :F1:INCPU1.SRC LARGE


                /**********      INCPU1.SRC     FILE     27 ØCT 8Ø      ***********/

                /* BEGIN INITIALIZE CPU1 MODULE */
   1            INIT$CPU1$MOD:  DO;

                /*****************************************************************/
                /*     LOCAL DATA DECLARTIONS                                    */
                /*****************************************************************/

   2   1          DECLARE LOG$CPU$ID BYTE,
                          ST$BTSTRP$ADR POINTER,
                          STAT$BTSTRP WORD,
                          PHYS$CPU$ID    LITERALLY    '17',
                          BOOTLOAD$CPU   LITERALLY    'Ø';

                /*****************************************************************/
                /*     EXTERNAL GLOBAL DATA DECLARATIONS                         */
                /*****************************************************************/

   3   1          DECLARE CPU$TBL$LOCK BYTE EXTERNAL,
                          LOG$CPU$NUM  BYTE EXTERNAL;

   4   1          DECLARE CPU$TABLE(8) STRUCTURE
                          (CPU$ID BYTE, CPU$ACK BYTE,
                           CPU$MAIL POINTER, CPU$TOTAL BYTE) EXTERNAL;
```

NON-MDS CONNECTED BOOTLOAD PROGRAM

Figure B-2

```
PL/M-86 COMPILER    INITCPU1MOD

                 $EJECT
                 /*****************************************************************/
                 /*     EXTERNAL ISBC 957A I/O SYSTEM PROCEDURES                  */
                 /*****************************************************************/

   5   1           LOAD: PROCEDURE(FILENAME,BIAS,SWITCH,ENTRY,STATUS) EXTERNAL;
   6   2              DECLARE (FILENAME,ENTRY,STATUS) POINTER;
   7   2              DECLARE (BIAS,SWITCH) WORD;
   8   2           END LOAD;

   9   1           EXIT: PROCEDURE EXTERNAL;
  10   2           END EXIT;

                 /*****************************************************************/
                 /*     LOCAL PROCEDURES                                          */
                 /*****************************************************************/

                 /*****************************************************************/
                 /*     CPU$WAIT                                                  */
                 /*---------------------------------------------------------------*/
                 /*     CAUSES THE NON-BOORLOAD CPU'S TO WAIT, IN A SPIN LOCK,    */
                 /*     UNTIL THE BOOTLOAD CPU SENDS THE ADDRESS OF THE BOOT-     */
                 /*     STRAP PROGRAM INDICATING THAT THIS PARTICULAR CPU CAN     */
                 /*     CONTINUE AND EXECUTE THE BOOTSTRAP PROGRAM.               */
                 /*****************************************************************/

  11   1           CPU$WAIT: PROCEDURE(WAIT$CPU$ID);

  12   2              DECLARE WAIT$CPU$ID BYTE,
                              BTSTRP$FLAG BYTE,
                              READY        LITERALLY '01',
                              NOT$READY    LITERALLY '00',
                              NULL          LITERALLY '00';
```

NON-MDS CONNECTED BOOTLOAD PROGRAM

Figure B-2 (cont'd)

```
                  $EJECT
                     /* INITIALIZE LOCK TO SPIN */
 13   2              BTSTRP$FLAG = NOT$READY;
 14   2              DO WHILE BTSTRP$FLAG <> READY;
 15   3                 DO WHILE LOCKSET(@CPU$TBL$LOCK,1);
 16   4                 END;
                        /* CHECK TO SEE IF CPU MAIL BOX HAS BEEN SET FROM NULL TO
                           BOOTSTRAP PROGRAM ADDRESS BY BOOTLOAD CPU */
 17   3                 IF CPU$TABLE(LOG$CPU$ID).CPU$MAIL <> NULL THEN
                           /* IF BOOTSTRAP ADDRESS IS IN MAIL BOX THEN SET READY
                              AND EXIT SPIN LOCK */
 18   3                    BTSTRP$FLAG = READY;
 19   3                 CPU$TBL$LOCK = 0;
 20   3                 END;      /* DO WHILE */
 21   2              RETURN;      /* END CPU$WAIT PROCEDURE */
 22   2              END CPU$WAIT;

                  /******************************************************************/
                  /*     BOOTLOAD$INTR                                              */
                  /*----------------------------------------------------------------*/
                  /*     ACTIVATING 'INTR' BUTTON ON 86/12 CHASSIS CAUSES JUMP      */
                  /*     TO THIS PROCEDURE. THE CPU$TABLE IS THEN ACCESSED INORDER*/
                  /*     TO PROVIDE A LOGICAL AND PHYSICAL ID FOR THIS CPU. IF      */
                  /*     THIS CPU BECOMES THE 'BOOTLOAD' CPU (LOGICALLY = 0) THEN */
                  /*     IT LOADS THE BOOTSTRAP PROGRAM AND JUMPS TO IT. OTHER-     */
                  /*     WISE IF ENTERS A WAIT STATE.                                */
                  /******************************************************************/

 23   1            BOOTLOAD$INTR: PROCEDURE INTERRUPT 62;

 24   2              DECLARE I BYTE;

                     /* CREATE DELAY TO ALLOW MDS CONNECTED CPU TO DEFAULT
                        TO BOOTLOAD CPU */
 25   2              DO I = 1 TO 100;
 26   3              END;
```

NON-MDS CONNECTED BOOTLOAD PROGRAM

Figure B-2 (cont'd)

```
                 $EJECT
 27   2              DO WHILE LOCKSET(@CPU$TBL$LOCK,1);
 28   3              END;
                     /* SET UNIQUE PHYSICAL CPU ID IN CPU$TABLE */
 29   2              CPU$TABLE(LOG$CPU$NUM).CPU$ID = PHYS$CPU$ID;
                     /* INCREMENT BOOTLOAD CPU COUNT */
 30   2              CPU$TABLE(BOOTLOAD$CPU).CPU$TOTAL = LOG$CPU$NUM + 1;
                     /* SET LOGICAL CPU ID */
 31   2              LOG$CPU$ID = LOG$CPU$NUM;
                     /* INCREMENT THE CPU NUMBER FOR THE NEXT CPU */
 32   2              LOG$CPU$NUM = LOG$CPU$NUM + 1;
 33   2              CPU$TBL$LOCK = 0;
                     /* IF CPU IS THE BOOTLOAD CPU */
 34   2              IF LOG$CPU$ID = 0 THEN
 35   2                 DO;
                        /* CREATE TIME DELAY TO ALLOW OTHER CPU'S TIME TO ACCESS
                           CPU$TABLE AND IDENTIFY THEMSELVES */
 36   3                 DO I = 1 TO 10;
 37   4                    CALL TIME(100);
 38   4                 END;
                        /* LOAD BOOTSTRAP PROGRAM INTO GLOBAL MEMORY */
 39   3                 CALL LOAD(@(':F1:BTSTRP '),0,0,@ST$BTSTRP$ADR,@STAT$BTSTRP);
 40   3              END;     /* IF */

                     /* IF NOT BOOTLOAD CPU (LOG$CPU$ID <> 0) */
                     ELSE
 41   2                 DO;
                        /* ENTER WAIT STATE BY EXECUTING SPIN LOCK */
 42   3                 CALL CPU$WAIT(LOG$CPU$ID);
                        /* SET BOOTSTRAP PROGRAM ADDRESS, THAT IS PASSED TO THE
                           CPU MAILBOX FROM THE BOOTLOAD CPU, SIGNALLING TIME
                           TO JUMP TO THE BOOTSTRAP PROGRAM */
 43   3                 ST$BTSTRP$ADR = CPU$TABLE(LOG$CPU$ID).CPU$MAIL;
 44   3                 END;     /* ELSE */
```

NON-MDS CONNECTED BOOTLOAD PROGRAM

Figure B-2 (cont'd)

```
PL/M-86 COMPILER    INITCPU1MOD


                $EJECT
                   /* JUMP TO BOOTSTRAP PROGRAM IN GLOBAL MEMORY */
 45   2            CALL ST$BTSTRP$ADR(LOG$CPU$ID,@CPU$TABLE,@CPU$TBL$LOCK);

 46   2          END BOOTLOAD$INTR;     /* END INTERRUPT PROCEDURE */

                 /* MAIN PROGRAM - CREATES INFINITE EXECUTION LOOP ONLY IN
                    THE BOOTLOAD CPU CONNECTED TO THE MDS. */

 47   1          DO WHILE 01;
 48   2          END;

 49   1        END;     /* INIT$CPU1$MOD */



MODULE INFORMATION:

     CODE AREA SIZE     = 01AFH    431D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0009H      9D
     MAXIMUM STACK SIZE = 0034H     52D
     141 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

NON-MDS CONNECTED BOOTLOAD PROGRAM

Figure B-2 (cont'd)

APPENDIX C. BOOTSTRAP PROGRAM LISTING

```
PL/M-86 COMPILER    BOOTSTRAPMOD

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE BOOTSTRAPMOD
OBJECT MODULE PLACED IN :F1:BTSTRP.OBJ
COMPILER INVOKED BY:  PLM86 :F1:BTSTRP.SRC LARGE


                 /**********     BTSTRP.SRC     FILE     20 NOV 80     ***********/

                 /* BEGIN BOOTSTRAP MODULE */
   1             BOOT$STRAP$MOD:  DO;

                 /*****************************************************************/
                 /*     LOCAL DATA DECLARATIONS                                   */
                 /*****************************************************************/

   2   1            DECLARE (KERNEL$AFTN,STATO$VAL,STATR$VAL,STATC$VAL,TRANS) WORD,
                            KERNEL$BUFFER(10000) BYTE;
   3   1            DECLARE (CS,IP,INDEX) WORD;
   4   1            DECLARE BT$STRP$DONE LITERALLY '01',
                            MAXTX  LITERALLY '4096';

                 /*****************************************************************/
                 /*     EXTERNAL ISBC 957A SYSTEM I/O PROCEDURES                  */
                 /*****************************************************************/

   5   1            OPEN: PROCEDURE(AFTN,FILE,ACCESS,ECHOAFTN,STATUS) EXTERNAL;
   6   2               DECLARE (AFTN,FILE,STATUS) POINTER;
   7   2               DECLARE (ACCESS,ECHOAFTN) WORD;
   8   2            END OPEN;
```

```
                  $EJECT
  9   1             READ: PROCEDURE(AFTN,BUFFER,COUNT,ACTUAL,STATUS) EXTERNAL;
 10   2                DECLARE (AFTN,COUNT) WORD;
 11   2                DECLARE (BUFFER,ACTUAL,STATUS) POINTER;
 12   2             END READ;

 13   1             CLOSE: PROCEDURE(AFTN,STATUS) EXTERNAL;
 14   2                DECLARE AFTN WORD;
 15   2                DECLARE STATUS POINTER;
 16   2             END CLOSE;

 17   1             EXIT: PROCEDURE EXTERNAL;
 18   2             END EXIT;

                  /******************************************************************/
                  /*     EXTERNAL PROCEDURES                                        */
                  /******************************************************************/

 19   1             READ$HEX$FILE: PROCEDURE(BUFF$PTR,CS$PTR,IP$PTR) EXTERNAL;
 20   2                DECLARE (BUFF$PTR,CS$PTR,IP$PTR) POINTER;

 21   2             END READ$HEX$FILE;

                  /******************************************************************/
                  /*     LOCAL PROCEDURES                                           */
                  /******************************************************************/

                  /******************************************************************/
                  /*     WAIT$CPU                                                   */
                  /*----------------------------------------------------------------*/
                  /*     CREATES WAIT STATE. IT SETS A SPIN LOCK UNTIL AN ACKNOW-   */
                  /*     LEDGE IS RECEIVED IN THE CPU'S MAIL BOX.                   */
                  /******************************************************************/
```

```
                $EJECT
 22   1            WAIT$CPU: PROCEDURE(WAIT$CPU$ID,CPU$TBL$PTR,TBL$LOCK$PTR) REENTRANT;

 23   2              DECLARE WAIT$CPU$ID BYTE,
                             CPU$TBL$PTR POINTER,
                             CPU$TABLE BASED CPU$TBL$PTR(8) STRUCTURE
                                       (CPU$ID BYTE, CPU$ACK BYTE,
                                        CPU$MAIL POINTER, CPU$TOT BYTE),
                             TBL$LOCK$PTR POINTER,
                             CPU$TBL$LOCK BASED TBL$LOCK$PTR BYTE;

 24   2              DECLARE CPU$FLAG BYTE,
                             DONE     LITERALLY '01',
                             NOT$DONE LITERALLY '00';

                     /* INITIALIZE SPIN-LOCK TO SPIN */
 25   2              CPU$FLAG = NOT$DONE;
 26   2              DO WHILE CPU$FLAG <> DONE;
 27   3                DO WHILE LOCKSET(@CPU$TBL$LOCK,1);
 28   4                END;
                       /* CHECK TO SEE IF CPU ACKNOWLEDGE FLAG IS SET */
 29   3                IF CPU$TABLE(WAIT$CPU$ID).CPU$ACK = BTSTRP$DONE THEN
                          /* IF ACKNOWLEDGE SET THEN SET CPU$FLAG TO DONE AND
                             EXIT SPIN-LOCK */
 30   3                   CPU$FLAG = DONE;
 31   3                CPU$TBL$LOCK = 0;
 32   3              END;     /* DO WHILE */
 33   2            END WAIT$CPU;     /* END WAIT$CPU PROCEDURE */
```

```
PL/M-86 COMPILER    BOOTSTRAPMOD

                    $EJECT
                    /*************************************************************/
                    /*      OUT$CHAR                                              */
                    /*------------------------------------------------------------*/
                    /*      UTILITY ROUTINE WHICH OUTPUTS CHARACTERS TO THE SERIAL */
                    /*      I/O PORT OF THE ISBC 86/12. USED TO PRINT DEBUG        */
                    /*      MESSAGES ON THE CRT.                                   */
                    /*************************************************************/
  34   1              OUT$CHAR: PROCEDURE(CHAR);
  35   2                 DECLARE CHAR BYTE;

  36   2                 DO WHILE(INPUT(ØDAH) AND Ø1H) = Ø; END;
  38   2                 OUTPUT(ØD8H) = CHAR;
  39   2              END OUT$CHAR;      /* END OUT$CHAR PROCEDURE */

                    /*************************************************************/
                    /*      OUT$WORD                                              */
                    /*------------------------------------------------------------*/
                    /*      UTILITY ROUTINE WHICH OUTPUTS NUMBERS TO THE SERIAL I/O */
                    /*      PORT OF THE ISBC 86/12. USED TO PRINT ERROR MESSAGE STATUS */
                    /*      VALUES.                                                */
                    /*************************************************************/
  4Ø   1              OUT$WORD: PROCEDURE(VALUE);
  41   2                 DECLARE VALUE WORD;
  42   2                 DECLARE ASCII(*) BYTE DATA ('Ø123456789ABCDEF');

  43   2                 CALL OUT$CHAR(ASCII(SHR(LOW(VALUE),4) AND ØFH));
  44   2                 CALL OUT$CHAR(ASCII(HIGH(VALUE) AND ØFH));
  45   2                 CALL OUT$CHAR(ASCII(SHR(HIGH(VALUE),4) AND ØFH));
  46   2                 CALL OUT$CHAR(ASCII(LOW(VALUE) AND ØFH));
  47   2              END OUT$WORD;      /* END OUT$WORD PROCEDURE */
```

```
                  $EJECT
                  /*********************************************************************/
                  /*     BTSTRP$ERROR                                                  */
                  /*-------------------------------------------------------------------*/
                  /*     OUTPUTS AN ERROR MESSAGE TO THE CRT IF THERE IS A PROBLEM     */
                  /*     WITH THE ISBC 957A SYSTEM I/O PROCEDURES.                     */
                  /*********************************************************************/

 48   1              BTSTRP$ERROR: PROCEDURE(STAT$VAL,IO$PROC) REENTRANT;
 49   2                 DECLARE STAT$VAL WORD,
                                IO$PROC  BYTE;
 50   2                 DECLARE Z        BYTE,
                                MSG$PTR POINTER,
                                MSG BASED MSG$PTR(1) BYTE,
                                MSG1(*) BYTE DATA('OPEN KERNEL FILE ERROR = '),
                                MSG2(*) BYTE DATA('READ KERNEL FILE ERROR = '),
                                MSG3(*) BYTE DATA('CLOSE KERNEL FILE ERROR = '),
                                CR   LITERALLY 'ØDH',
                                LF   LITERALLY 'ØAH';

                        /* SELECT APPROPRIATE ERROR MESSAGE */
 51   2                 DO CASE IO$PROC;
 52   3                    MSG$PTR = @MSG1;
 53   3                    MSG$PTR = @MSG2;
 54   3                    MSG$PTR = @MSG3;
 55   3                 END;
                        /* OUTPUT ERROR MESSAGE TO CRT */
 56   2                 DO Z = Ø TO 29;
 57   3                    CALL OUT$CHAR(MSG(Z));
 58   3                 END;
                        /* OUTPUT STATUS VALUE TO CRT */
 59   2                 CALL OUT$WORD(STAT$VAL);
 6Ø   2              END BTSTRP$ERROR;     /* END BTSTRP$ERROR PROCEDURE */
```

```
                  $EJECT
                  /*****************************************************************/
                  /*     BOOT$STRAP                                                */
                  /*---------------------------------------------------------------*/
                  /*     THE BOOTLOAD CPU (LOGICALLY = Ø) LOADS THE KERNEL FILE    */
                  /*     INTO A GLOBAL MEMORY BUFFER. EACH CPU, IN TURN, (AS CON-  */
                  /*     TROLLED BY THE BOOTLOAD CPU) EXECUTES THIS GLOBAL BOOT-   */
                  /*     STRAP PROGRAM TO LOAD THAT PARTICULAR CPU'S LOCAL MEMORY. */
                  /*****************************************************************/

  61   1            BOOT$STRAP: PROCEDURE(LOG$CPU$ID,CPU$TBI$PTR,TBL$LOCK$PTR) PUBLIC REENTRANT;

  62   2              DECLARE LOG$CPU$ID BYTE,
                              CPU$TBL$PTR POINTER,
                              CPU$TABLE BASED CPU$TBL$PTR(8) STRUCTURE
                                        (CPU$ID BYTE, CPU$ACK BYTE,
                                         CPU$MAIL PCINTER, CPU$TOTAL BYTE),
                              TBL$LOCK$PTR POINTER,
                              CPU$TBI$LOCK BASED TBL$LOCK$PTR BYTE;

  63   2              DECLARE MEM$KCSIP$PTR POINTER,
                              KCSIP STRUCTURE (OFF WORD, SEG WORD)
                              AT (@MEM$KCSIP$PTR);

  64   2              DECLARE (Z,TOTAL$CPUS,LOG$CPU$NUM) BYTE,
                              ENDMSG(*) BYTE DATA('BOOTSTRAP COMPLETE ');

                      /* IF THIS IS BOOTLOAD CPU */
  65   2              IF LOG$CPU$ID = Ø THEN
  66   2                DO;
```

```
                  $EJECT
                             /* OPEN KERNEL HEX FILE */
  67   3                     CALL OPEN(@KERNEL$AFTN,@(':F1:KERNEL.HEX '),1,Ø,@STATO$VAL);
  68   3                     IF STATO$VAL <> Ø THEN
  69   3                        CALL BTSTRP$ERROR(STATO$VAL,Ø);
                             /* READ KERNAL HEX FILE INTO GLOBAL MEMORY BUFFER */
  7Ø   3                     CALL READ(KERNEL$AFTN,@KERNEL$BUFFER,MAXTX,@TRANS,@STATR$VAL);
  71   3                     IF STATR$VAL <> Ø THEN
  72   3                        CALL BTSTRP$ERROR(STATR$VAL,1);
                             /* INCREMENT INDEX TO NEXT EMPTY KERNEL$BUFFER ADDRESS */
  73   3                     INDEX = MAXTX + 1;
                             /* READ KERNEL HEX FILE INTO GLOBAL MEMORY UNTIL BYTES
                                TRANSFERRED IS LESS THAN 4Ø96 BYTES; INDICATING EOF */
  74   3                     DO WHILE TRANS = MAXTX;
  75   4                        CALL READ(KERNEL$AFTN,@KERNEL$BUFFER(INDEX),MAXTX,@TRANS,@STATR$VAL);
  76   4                        IF STATR$VAL <> Ø THEN
  77   4                           CALL BTSTRP$ERROR(STATR$VAL,1);
                                /* INCREMENT INDEX TO NEXT EMPTY KERNEL$BUFFER ADDRESS */
  78   4                        INDEX = INDEX + MAXTX + 1;
  79   4                     END;     /* DO WHILE */
                             /* CLOSE KERNEL HEX FILE */
  8Ø   3                     CALL CLOSE(KERNEL$AFTN,@STATC$VAL);
  81   3                     IF STATC$VAL <> Ø THEN
  82   3                        CALL BTSTRP$ERROR(STATC$VAL,2);
                             /* LOAD THE KERNEL HEX FILE IN GLOBAL MEMORY INTO LOCAL
                                MEMORY OF THE BOOTLOAD CPU */
  83   3                     CALL READ$HEX$FILE(@KERNEL$BUFFER,@CS,@IP);
  84   3                     DO WHILE LOCKSET(@CPU$TBL$LOCK,1);
  85   4                     END;
                             /* DETERMINE THE NUMBER OF REMAINING CPU'S TO BE LOADED
                                FROM BOOTLOAD CPU TOTAL AND SUBTRACT ONE SO BOOTLOAD
                                CPU ISN'T COUNTED */
  86   3                     TOTAL$CPUS = CPU$TABLE(Ø).CPU$TOTAL - 1;
  87   3                     CPU$TBL$LOCK = Ø;
```

```
                $EJECT
                             /* LOAD THE KERNEL HEX FILE INTO LOCAL MEMORY OF REMAINING
                                CPU'S */
  88   3                     DO LOG$CPU$NUM = 1 TO TOTAL$CPUS;
  89   4                        DO WHILE LOCKSET(@CPU$TBL$LOCK,1);
  90   5                        END;
                                /* SET FLAG TO ALLOW CPU TO ENTER THE BOOSTRAP PROGRAM */
  91   4                        CPU$TABLE(LOG$CPU$NUM).CPU$MAIL = @BOOT$STRAP;
  92   4                        CPU$TBL$LOCK = 0;
                                /* ENTER SPIN-LOCK UNTIL THE OTHER CPU IS DONE LOADING
                                   THE KERNEL HEX FILE */
  93   4                        CALL WAIT$CPU(LOG$CPU$NUM,@CPU$TABLE,@CPU$TBL$LOCK);
  94   4                     END;     /* DO */
                             /* AFTER ALL CPU'S HAVE LOADED THE KERNEL HEX FILE SET
                                ACKNOWLEDGE FLAG TO ALLOW ALL CPU'S TO JUMP TO THE
                                KERNEL FILE AND EXECUTE */
  95   3                     CPU$TABLE(0).CPU$ACK = BTSTRP$DONE;
  96   3                  END;     /* IF */
                          /* IF NOT BOOTLOAD CPU */
                          ELSE
  97   2                     DO;
                             /* LOAD KERNEL HEX FILE IN GLOBAL MEMORY INTO LOCAL MEMORY
                                OF THIS PARTICULAR CPU */
  98   3                     CALL READ$HEX$FILE(@KERNEL$BUFFER,@CS,@IP);
  99   3                     DO WHILE LOCKSET(@CPU$TBL$LOCK,1);
 100   4                     END;
                             /* INDICATE TO BOOTLOAD CPU THAT LOADING IS COMPLETE */
 101   3                     CPU$TABLE(LOG$CPU$ID).CPU$ACK = BTSTRP$DONE;
 102   3                     CPU$TBL$LOCK = 0;
                             /* ENTER SPIN-LOCK UNTIL ALL OTHER CPU'S ARE DONE LOADING
                                KERNEL HEX FILE */
 103   3                     CALL WAIT$CPU(0,@CPU$TABLE,@CPU$TBL$LOCK);
 104   3                  END;     /* ELSE */
```

```
PL/M-86 COMPILER    BOOTSTRAPMOD


                    $EJECT
                          /* OUTPUT BOOTSTRAP COMPLETE MESSAGE TO CRT */
 105   2                  DO Z = 0 TO 18;
 106   3                     CALL OUT$CHAR(ENDMSG(Z));
 107   3                  END;

                          /* SET CS AND IP, FOR KERNEL PROGRAM, WHICH WERE EXTRACTED
                             FROM THE KERNEL HEX FILE IN READ$HEX$FILE. THESE ARE
                             USED TO BUILD THE POINTER MEM$KCSIP$PTR. */
 108   2                  KCSIP.SEG = CS;
 109   2                  KCSIP.OFF = IP;
                          /* JUMP TO KERNEL PROGRAM */
 110   2                  CALL MEM$KCSIP$PTR(LOG$CPU$ID,CPU$TABLE(LOG$CPU$ID).CPU$ID);

 111   2               END BOOT$STRAP;     /* END BOOT$STRAP PROCEDURE */

 112   1            END;     /* END BOOT$STRAP$MOD */


MODULE INFORMATION:

     CODE AREA SIZE     = 03E5H     997D
     CONSTANT AREA SIZE = 0000H       0D
     VARIABLE AREA SIZE = 2720H   10016D
     MAXIMUM STACK SIZE = 002EH      46D
     264 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
PL/M-86 COMPILER    READHEXMOD

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE READHEXMOD
OBJECT MODULE PLACED IN :F1:RDHEX.OBJ
COMPILER INVOKED BY:  PLM86 :F1:RDHEX.SRC LARGE


              /**********     RDHEX.SRC     FILE     20 NOV 80     *************/

              /* BEGIN READ HEX MODULE */
  1           READ$HEX$MOD:  DO;

  2   1           OUT$CHAR: PROCEDURE(J);
  3   2               DECLARE J BYTE;
  4   2               DO WHILE(INPUT(0DAH) AND 01H) = 0; END;
  6   2               OUTPUT(0D8H) = J;
  7   2           END OUT$CHAR;

  8   1           ERROR: PROCEDURE;
  9   2              DECLARE X BYTE;
 10   2              DECLARE HEXMSG(*) BYTE DATA('READ$HEX$FILE$ ERROR ');
 11   2              DO X = 0 TO 19;
 12   3                 CALL OUT$CHAR(HEXMSG(X));
 13   3              END;
 14   2           END ERROR;
```

```
                  $EJECT
                  /*****************************************************************/
                  /*     READ$HEX$FILE                                             */
                  /*---------------------------------------------------------------*/
                  /* THIS PROCEDURE IS PART OF THE BOOTSTRAP PROGRAM. IT READS A   */
                  /* HEXADECIMAL FILE LOCATED IN A GLOBAL MEMORY BUFFER AND        */
                  /* RELOCATES THE HEX FILE IN LOCAL MEMEORY ACCORDING TO THE      */
                  /* ADDRESSES SPECIFIED IN THE HEX FILE. SIMULTANEOUSLY READ$HEX$ */
                  /* FILE CHANGES THE HEXADECIMAL FILE TO A BINARY OBJECT FILE     */
                  /* AS IT READS THE HEX FILE IN THE GLOBAL MEMORY BUFFER.         */
                  /*****************************************************************/

 15    1            READ$HEX$FILE: PROCEDURE(BUFF$PTR,CS$PTR,IP$PTR) PUBLIC;

 16    2             DECLARE BUFF$PTR POINTER,
                             BUFFER BASED BUFF$PTR(10000) BYTE;
 17    2             DECLARE CS$PTR POINTER,
                             CS BASED CS$PTR WORD,
                             IP$PTR POINTER,
                             IP BASED IP$PTR WORD;

 18    2             DECLARE MEM$ARG1$PTR POINTER,
                         ARG1 STRUCTURE (OFF WORD, SEG WORD)
                             AT (@MEM$ARG1$PTR),
                         MEM$ARG1 BASED MEM$ARG1$PTR BYTE,
                         MEM$WORD$ARG1 BASED MEM$ARG1$PTR WORD;

 19    2             DECLARE (REC$TYPE,LEN) BYTE,
                         (CHECK$SUM, T, K) BYTE,
                         HEX$FLAG  BYTE,
                         OFFSET WORD,
                             I WORD;
```

```
                $EJECT
 20   2                   DECLARE HEX$NOT$DONE LITERALLY '00',
                              HEX$DONE     LITERALLY '01';

 21   2              DECLARE HMSG(*) BYTE DATA('ENTERED READ$HEX     ');
 22   2              DECLARE J BYTE;


 23   2              CHG$HEX: PROCEDURE(C) WORD;
                     /* THIS ROUTINE CONVERTS THE INPUT PARAMETER FROM ASCII TO ITS
                        BINARY EQUIVALENT AND RETURNS IS AS THE VALUE OF THE PROCEDURE
                        NO CHECK IS MADE FOR INPUT VALIDITY. */
 24   3                 DECLARE C BYTE;
 25   3                 IF C <= '9' THEN RETURN DOUBLE(C - 30H);
 27   3                 ELSE RETURN DOUBLE (C - 37H);
 28   3              END CHG$HEX;

 29   2              READ$CHAR: PROCEDURE BYTE;
                     /* THIS ROUTINE READS A BYTE FROM AN ARRAY BUFFER OF BYTES AND
                        RETURNS THE BYTE. */
 30   3                 DECLARE CHAR BYTE;
 31   3                 CHAR = BUFFER(I);
 32   3                     I = I + 1;
 33   3                 RETURN CHAR;
 34   3              END READ$CHAR;

 35   2              READ$BYTE: PROCEDURE BYTE;
                     /* THIS ROUTINE READS TWO HEX BYTES AND RETURNS THEIR BINARY
                        BYTE VALUE */
 36   3                 DECLARE T BYTE;
 37   3                 T = LOW(CHG$HEX(READ$CHAR));
 38   3                 T = SHL(T,4) + LOW(CHG$HEX(READ$CHAR));
 39   3                 CHECK$SUM = CHECK$SUM + T;
 40   3                 RETURN T;
 41   3              END READ$BYTE;
```

```
                   $EJECT
 42   2              READ$WORD: PROCEDURE WORD;
                     /* THIS ROUTINE READS FOUR HEX BYTES AND RETURNS THEIR BINARY
                        WORD VALUE. */
 43   3                 DECLARE T BYTE;
 44   3                 T = READ$BYTE;
 45   3                 RETURN SHL(DOUBLE(T),8) + DOUBLE(READ$BYTE);
 46   3              END READ$WORD;

                   /*************** BEGIN READ$HEX$FILE MAIN PROGRAM ***************/

                     /* INITIALIZE BUFFER INDEX AND HEX$FLAG */
 47   2              I = 0; HEX$FLAG = HEX$NOT$DONE;
                     /* READ HEX FILE UNTIL EOF */
 49   2              DO WHILE HEX$FLAG <> HEX$DONE;
 50   3                 DO WHILE READ$CHAR <> ':';
 51   4                 END;
                        /* ESTABLISH LENGTH,OFFSET AND RECORD TYPE */
 52   3                 LEN = READ$BYTE;
 53   3                 OFFSET = READ$WORD;
 54   3                 ARG1.OFF = OFFSET;
 55   3                 REC$TYPE = READ$BYTE;
                        /* IF START ADDRESS RECORD */
 56   3                 IF REC$TYPE = 03 THEN
 57   3                    DO;
 58   4                    CS = READ$WORD;
 59   4                    IP = READ$WORD;
 60   4                    END;
                        /* IF ADDRESS RECORD */
 61   3                 IF REC$TYPE = 02 THEN
 62   3                    ARG1.SEG = READ$WORD;
                        /* IF EOF RECORD */
 63   3                 IF REC$TYPE = 01 THEN
 64   3                    IF OFFSET <> 0 THEN IP = OFFSET;
```

```
PL/M-86 COMPILER    READHEXMOD

                   $EJECT
                          /* IF DATA RECORD */
 66   3                   IF REC$TYPE = 00 THEN
                             /* READ RECORD DATA INTO BUFFER */
 67   3                      DO K = 1 TO LEN;
 68   4                        T,MEM$ARG1 = READ$BYTE;
 69   4                            IF MEM$ARG1 <> T THEN CALL ERROR;
 71   4                        ARG1.OFF = ARG1.OFF + 1;
 72   4                      END;
 73   3                   T = READ$BYTE;
 74   3                   IF CHECK$SUM <> 0 THEN CALL ERROR;
                          /* IF END OF FILE */
 76   3                   IF REC$TYPE = 01 AND LEN = 0 THEN
 77   3                      HEX$FLAG = HEX$DONE;
 78   3                END;  /* DO WHILE */

 79   2            END READ$HEX$FILE; /*  END PROCEDURE  */

 80   1            END;    /* READ$HEX$MOD */


MODULE INFORMATION:

     CODE AREA SIZE     = 0232H    562D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 001FH     31D
     MAXIMUM STACK SIZE = 0014H     20D
     139 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

APPENDIX D. KERNEL LOADER LISTING

```
/*************************************************************/
/*     Kernel Loader Routine                                 */
/*-----------------------------------------------------------*/
/*      This pseudo-code is included to familiarize the      */
/* reader with the kernel loader routine function and is not */
/* tested code.                                              */
/*************************************************************/

KERNEL$LOADER: PROCEDURE;

   /* SUBROUTINE TO REINITIALIZE THE APPLICATION PROCESS */
   REINITIALIZE: PROCEDURE(PROC$NUM);

      /* REINITIALIZE THE ADDRESS SPACE INDEX (ASI) */
      ASI = Ø;
      /* INDEX THROUGH THE PROCESS ADDRESS SPACE (PAS) TO
         RELOAD EACH SEGMENT */
      DO WHILE(PDT(PROC$NUM).PAS(ASI)<>NULL)OR(ASI <> MAX$SEG));
        /* RELOAD THE SEGMENT */
        SEG$LOC = SWAP$IN(PDT(PROC$NUM).PAS(ASI));
        /* RECORD SEGMENT LOCATION IN THE PROCESS PARAMETER
          BLOCK */
        PPB(ASI) = SEG$LOC;
        /* INCREMENT THE ADDRESS SPACE INDEX */
        ASI = ASI + 1;
      END;  /* DO WHILE */
      /* CREATE PROCESS DESCRIPTOR SEGMENT */
      CALL CREATE$PROCESS(@PPB);

   END;  /* REINITIALIZE PROCEDURE */


   /* REINITIALIZE CPU EVENTCOUNT AWAITED VALUE */
   AWAIT$VALUE = 1;

   /* ENTER DO FOREVER LOOP */
   DO WHILE Ø1;

     /* CHECK TO SEE IF THIS IS THE LOAD CPU */
     IF LOG$CPU$ID = Ø THEN DO;

       /* REINITIALIZE THE LOAD CPU EVENTCOUNT VALUE AWAITED */
       CPU$AWAIT$VALUE = 1;
       /* DETERMINE THE NUMBER OF CPUS AVAILABLE FOR RECOVERY
          FROM THE LOAD CPU ENTRY IN THE CONFIGURATION TABLE */
      TOTAL$CPUS = CONFIG$TABLE(0).CPU$TOTAL;
      /* INDEX THROUGH THE PDT TO REINITIALIZE ALL PROCESSES */
      DO PROC$NUM = 0 TO MAX$PROC;

        /* DETERMINE PROCESS CPU AFFINITY */
        PROC$AFFINITY = PDT(PROC$NUM).PCM(TOTAL$CPUS);
        /* IF THE AFFINTIY IS FOR THE LOAD CPU THEN DO */
        IF PROC$AFFINITY = 0 THEN
           /* REINITIALIZE THE APPLICATION PROCESS */
           CALL RINITIALIZE(PROC$NUM);

        /* IF NOT THE LOAD CPU AFFINITY THEN */
        ELSE DO;

          /* SIGNAL THE TARGET CPU LOADER PROCESS */
          CALL ADVANCE(SYS$EVC$TBL(PROC$AFFINITY));
          /* ENTER A WAIT STATE UNTIL THE TARGET CPU HAS
             COMPLETED THE PROCESS REINITIALIZATION */
          CALL AWAIT(SYS$EVC$TBL(0),CPU0$AWAIT$VALUE);
          /* INCREMENT EVENTCOUNT VALUE AWAITED */
          CPU0$AWAIT$VALUE = CPU0$AWAIT$VALUE + 1;

        END; /* ELSE */


      END; /* DO */

      /* RESTART THE SYSTEM */
      CALL ADVANCE(SYS$EVC$TBL(START$EVENT));
      /* ENTER A WAIT STATE UNTIL RESTARTED */
      CALL AWAIT(SYS$EVC$TBL(0),CPU0$AWAIT$VALUE);

     END;  /* IF LOG$CPU$ID = 0 */

    /* IF NOT THE LOAD CPU THEN FOLLOW THESE INSTRUCTIONS */
    ELSE DO;

      /* ENTER A WAIT STATE UNTIL SIGNALLED BY THE LOAD CPU
         TO RELOAD A PROCESS */
      CALL AWAIT(SYS$EVC$TBL(LOG$CPU$ID),AWAIT$VALUE);
      /* INCREMENT THE EVENTCOUNT VALUE AWAITED */
      AWAIT$VALUE = AWAIT$VALUE + 1;
      /* REINITIALIZE THE APPLICATION PROCESS */
      CALL REINITIALIZE(PROC$NUM);

    END; /* ELSE */

  END; /* DO FOREVER */

END; /* KERNEL$LOADER PROCEDURE */
```

LIST OF REFERENCES

1. Avizienis, A., "Fault-Tolerance: The Survival Attribute of Digital Systems", Proceedings of the IEEE, Vol. 66, No. 10, pp. 1109-1125, October 1978.

2. Brenner, R., Multiple Microprocessor Architecture for Smart Sensor Focal Plane Image Processing, M.S. Thesis, Naval Postgraduate school, June 1980.

3. Hopkins, A.L. Jr. etal, "FTMP- A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", Proceedings of the IEEE, Vol. 66, No. 10, pp. 1221-1239, October 1978.

4. Intel Corporation, The 8086 Family User's Manual, 1979.

5. Intel Corporation, PL/M-86 Programming Manual, 1979.

6. Intel Corporation, MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users, 1979.

7. Intel Corporation, ISIS-II PL/M-86 Compiler Operator's Manual, 1979.

8. Intel Corporation, MCS-86 Macro Assembler Operating Instructions for ISIS-II Users, 1979.

9. Intel Corporation, iSBC 975A-iSBC 86/12A Interface and Execution Package Manual, 1979.

10. Intel Corporation, iCS 80 Industrial Chassis Hardware Reference Manual, 1979.

11. Intel Corporation, iSBC 86/12A Single Board Computer Hardware Reference Manual, 1979.

12. Katsuki, D. etal, "Pluribus-An Operational Fault-Tolerant

Multiprocessor", Proceedings of the IEEE, Vol. 66, No. 10, pp. 1146-1159, October 1978.

13. Luniewski, A., A Simple and Flexible System Initialization Mechanism, M.S. Thesis, M.I.T., May 1977.

14. Moore, E.E. and Gary, A.V., The Design and Implementation of the Memory Manager for a Secure Archival Storage System, M.S. Thesis, Naval Postgraduate School, June 1980.

15. O'Connell, J., and Richardson, D., Secure Design for a Multi-Processor Operating System, M.S. Thesis, Naval Postgraduate School, June 1980.

16. Organik, S., Multics: An Examination of It's Structure, M.I.T. Press, 1972.

17. Rapantzikos, D., Implementation of a Distributed Multiple Microcomputer Operating System, M.S. thesis in preparation Naval Postgraduate School, (expected completion, April 1981).

18. Reed, D.P., Processor Multiplexing in a Layered Operating System, M.S. Thesis. M.I.T., 1976.

19. Rennels, D.A., "Distributed Fault-Tolerant Computer Systems", Computer, pp. 55-65, March 1980.

20. Ross, J.I., Design of a System Initialization Mechanism for a Multiple Microcomputer , M.S. Thesis, Naval Postgraduate School, June 1980.

21. Schell, R.R., Dynamic Reconfiguration in a Modular Computer System, Ph.D. Thesis, M.I.T., May 1971.

22. Schell, R.R., Kodres, U.R., Amir, H., Wasson, J. and Tao, T.F., Processing of Infrared Images by Multiple Microcomputer System, Proceedings of the SPIE, Vol. 241, 1980.

23. Wasson, W.J., Detailed Design of the Kernel of a Real Time Multiprocessor Operating System, M.S. Thesis, Naval Postgraduate School, June 1980.

24. Wensley, J.H., etal, "Sift: Design and Analysis of a Fault Tolerant Computer for Aircraft Control", Proceedings of the IEEE, Vol. 66, No. 10, pp. 1240-1255, October 1978.

25. Verhofstad, J.S.M., "Recovery Techniques for Database Systems", ACM Computing Surveys, Vol. 10, No. 2, pp 167-195, June 1978.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center — 2
Cameron Station
Alexandria, Virginia 22314

2. Library, Code 0142 — 2
Naval Postgraduate School
Monterey, California 93940

3. Department Chairman, Code 52 — 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

4. Col. R. R. Schell, Code 52Sj — 4
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

5. Asst. Professor U. R. Kodres, Code 52Kr — 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

6. Professor T. F. Tao, Code 62Tv — 3
Department of Electrical Engineering
Naval Postgraduate School
Monterey, California 93940

7. Demosthenis Rapantzikos — 1
Karaoli 7
Salamis
Nisos Salamis
Greece

8. Intel Corporation — 1
Attn: Mr. Robert Childs
Mail Code: SC4-490
3065 Bowers Avenue
Santa Clara, California 95051

9. Lt Richard L. Anderson, USN — 3
Commander Naval Military Personnel Command
(NMPC-16F1)
Washington, D. C. 20370